Analyzing Demand in Non-Strict Functional Programming Languages

Dissertation
zur Erlangung des Doktorgrades
der Naturwissenschaften

vorgelegt beim Fachbereich Biologie und Informatik
der Johann Wolfgang Goethe - Universität
in Frankfurt am Main

von
Marko Schütz
aus Frankfurt

Frankfurt 2000
(DF1)

Vom Fachbereich Informatik der Johann Wolfgang Goeth
Universität als Dissertation angenommen.

Dekan:	Prof. Dr. Karl-Dieter Entian
Gutachter:	Prof. Dr. Manfred Schmidt-Schauß Johann Wolfgang Goethe-Universität Frankfurt Fachbereich Biologie und Informatik Institut für Informatik Robert-Mayer-Str. 11-15 **D-60325** Frankfurt am Main
	Prof. Dr. Elfriede Fehr Freie Universität Berlin Fachbereich Mathematik und Informatik, Institut für Informatik Takustr. 9 **D-14195** Berlin

Datum der Disputation: 5. Juli 2001

Acknowledgments

It is my pleasure to thank all those who directly or indirectly supported my work on this dissertation.

The research on which this dissertation reports has been conducted during the years 1995 to 2000 while employed at Freie Universität, Berlin and at J.W.Goethe-Universität, Frankfurt.

First and foremost I thank Manfred Schmidt-Schauß for the uncounted enlightening talks we had during this time. The strong interest in the progress of this work he took at all times, his valuable comments and his constructive criticism, expressed in a straightforward but friendly manner, not only helped me gain new insights but also motivated me time and time again. I am glad to have worked with him and grateful for having been his student.

In love I thank Petra Schmuck for her support, her assistance, and the diverse "distractions" from this work that helped me find the necessary distance and reinvigoration.

During the work on their respective master's theses, Dirk Rehberger, Matthias Mann and Norbert Klose all contributed to the current state of this work with questions, suggestions, discussion, implementation work, and practical trials. Matthias Mann continued his contribution even after this time by reading and commenting on drafts.

The members of the colloquium of the professorship for artificial intelligence and software technology provided inspiring ideas from related areas.

Eugen Seidel repeatedly gave helpful advice improving the phrasing of this dissertation.

Contents

1 Introduction

Often programs behave unexpectedly. This kind of behavior can be an unexpected result, failure to terminate, or termination due to some exceptional state within the program. In all these cases, there is disagreement between the programmer's mental model of the program's behavior and the observed behavior. Depending on the exact case, one of the first questions a programmer will ask himself is "Why is this result computed?", "Why is no result computed?", or some such question, in an effort to find out which to modify: the program or his mental model. A second question arising as a consequence of the first is "What are the values for which the function exhibits this unexpected behavior?" or "For which values will the function fail to terminate?". These are precisely the questions the present analysis sets out to answer and which it will be able to answer in many cases. But our analysis goes much further, since the classes of results, the *demands* in the analysis, are not restricted to being data values, but may represent non-termination as well as higher-order values. Thus we may indeed formulate an input to the analysis asking: "For which values is the partial application of + to only its first argument a function which will not terminate for any value given as its missing argument?". Even if such a question seems confusing when stated verbally, it can be succinctly stated in the demand analysis framework as `plus` $x \in \texttt{Top} \overset{\forall}{\rightarrow} \texttt{Bot}$. Higher-order functions are nothing special in functional programming, so questions like

the above will naturally arise and this is one of the most simple examples involving higher-order functions.

Demand analysis can analyze a wide variety of semantic properties and thus these properties could be checked, similarly to the type check in modern functional languages, where a type is inferred and then compared to the programmer's type annotation. For example, a programmer could state that a function needs to evaluate its argument strictly if one of its applications needs to be evaluated strictly, and be warned if the function's argument is inferred to need spine-strict evaluation in this case.

While helping improve agreement between the intended semantics and the semantics actually implemented is an important application for demand analysis, another is the improvement of program efficiency. Here the application of optimizations is sometimes guarded by semantic properties, which demand analysis can automatically check ensuring safety of the optimization.

Demands represent sets of expressions. The demand definition $\texttt{Inf} = \langle \texttt{Bot}, \texttt{Top} : \texttt{Inf} \rangle$ for example describes expressions on which the `length` function does not terminate. Finitely structured elements represented by `Inf` are the expressions evaluating to a finite spine of `:`-constructors ending in a non-terminating expression. The infinite lists come in as least upper bounds of ascending chains of finite elements.

Demand definitions are constructed as needed. We assume that initially only one demand definition together with some demand constants are known. New demand definitions may be added as the results of analyses and may then be used to form initial constraints for new analyses.

Demands can be interpreted as types. However, they do not directly correspond to the polymorphic types like `List a` in Haskell or α `list` in ML. In the example above the elements of the list can be taken from any program type, they may even be untyped, whereas a member of type `List a` will need to have its elements taken from the type the variable `a` is instantiated with. Demands cannot be parameterized with other demands. However, all the monomorphic instances of a polymorphic type actually used in a program can very well be represented as demands. Examples are $\texttt{ListBool} = \langle \texttt{[]}, \texttt{Bool} : \texttt{ListBool} \rangle$ for the finite lists containing boolean values or $\texttt{InfListBool} = \langle \texttt{Bot}, \texttt{ListBool} : \texttt{InfListBool} \rangle$ for streams with finite lists of boolean values as elements.

Demand analysis is a backward analysis. Historically, the terms *forward* and *backward* are related to the information flow in the analysis, where the flow of information from the entire expression to its sub-expressions is seen as the backward direction and the flow of information from the sub-expressions to the entire expression is seen as forward [Hug88].

Applications of demand analysis include strictness analysis, a more general analysis of evaluation degrees, absence analysis, exception analysis, and termination analysis. Strictness analysis, for example, can be performed by analyzing e.g. $\texttt{length}\ xs \in \texttt{Bot}$ resulting in `Inf` and since the expressions specified by `Inf` include those specified by `Bot` the result implies that `length` is strict.

1.1 Overview

The main contribution of this dissertation is

- the definition of a highly precise notion of demand based entirely on *operational* semantics of the underlying language and

- the introduction of successful demand-analysis calculi that are in some cases able to employ induction and that are in some cases able to form upper bounds.

The dissertation is structured as follows. Chapter 2 introduces the language Λ, a λ-calculus extended with constructors and case-expressions for their distinction. We give Λ a non-strict operational semantics by defining a normal-order reduction for Λ. Due to the presence of constructors our notion of WHNF is more diverse than that of the λ-calculus. We distinguish between WHNFs that are completely applied constructor expressions and those that are either incompletely applied constructor expressions or that are abstractions. Reductions terminate in WHNFs. Reduction and normal-order reduction have important properties that distinguish them as an adequate choice.

1. The result obtained by some reduction sequence can also be obtained by a reduction sequence in which all normal-order reduction occurs before reduction not in normal order. We say reduction sequences can be *standardized*.

2. Reduction of an expression has no effect on the existence of a terminating reduction sequence. This property is called *invariance of termination*.

As semantics of the Λ-expressions we choose the equivalence classes of the contextual equivalence. Two expressions are contextually equivalent if there is no context, i.e. no expression with a hole, for which insertion of each of the expressions results in differently terminating expressions. This equivalence is more general than the reflexive, symmetric, and transitive closure of the reduction relation. Since the definition of contextual order, which is used to define contextual equivalence, uses quantification over all possible contexts, it is often unwieldy for proofs. With the context lemmata we obtain the possibility to narrow our attention to some specific set of contexts. This aids in shortening the proofs and in making them more comprehensible.

Following the definition of least upper bounds and the proof of continuity for arbitrary contexts we state that Turing's $\boldsymbol{\theta}$ is a least-fixpoint combinator and that $\boldsymbol{\theta}\ t$ is equivalent to the least upper bound of the ascending chain consisting of i-fold iterated t's.

Closing chapter 2 we note how super-combinators can be translated into Λ and state that Λ does not form a CPO with the contextual order.

The purpose of our demand analysis calculi is to compute bindings for free variables of a Λ-expression under which the expression reduces to a specified result. We specify sets of such results using demands which are introduced and investigated in chapter 3. Demands stand for sets of Λ-expressions and e.g. provide syntactic constructs for the representation of union and intersection. They are assigned meaning in a two-phased process: first the representation of a demand is defined using the smallest fixpoint of a recursive function, and then their concretization is defined as the closure with respect to forming contextually least upper bounds of ascending chains from the representation. Representation as well as concretization are closed with respect to contextual equivalence and therefore also with respect to reduction. On the other hand, demands can very precisely specify which Λ-expressions should be included, e.g. for every constructor normal form there is a demand representing exactly those Λ-expressions contextually equivalent to this constructor normal form. In chapter 5 demands are ex-

tended to provide a similar precision for higher-order expressions.

All the expressions from the representation of a demand can be found in that demand's concretization. Additionally, expressions that may e.g. be evaluated to arbitrary depth are introduced into a demand's concretization. Such expressions cannot be members of the representation of any demand.

Demands have the same expressive power as Turing-machines. This holds even for a restricted class of demands. The equipotence proof is in two directions: for one we reduce Turing-machines to demands, the other amounts to giving an enumeration method for demands.

Concretizations are closed with respect to forming contextually least upper bounds of their ascending chains. This is an important technical result, particularly for the soundness and completeness proofs, because it will allow to conclude that a statement holds for the solutions from a concretization if that statement holds for the solutions from the corresponding representation.

In section 3.6 we interpret demand definitions as monotonous functions and observe that the concretization of some is the least fixpoint of that function, for others it is the greatest fixpoint of that function, and for yet others it is a fixpoint of that function, but neither the least nor the greatest.

Chapter 3 closes with a section on demand transformations. These replace syntactically complex demands by (subjectively) simpler demands while maintaining the concretization.

Chapter 4 presents demand analysis and introduces two calculi for performing this analysis. Demand analysis receives a constraint $s \in D$ as input in which s is an open Λ-expression and D is a demand, and tries to compute bindings for the free variables for which the Λ-expression belongs to D's concretization. Both calculi generate a demand expression in order to specify the set of solutions and they only differ in their rule set: one, ADE, may use additional rules not available to the other, CADE. CADE is sound and complete if it terminates whereas ADE is only sound, but ADE terminates for more inputs than CADE.

We present the calculi as non-deterministic rule sets transforming a tree-like data structure into another by appending new leaves below the leaves in the former. The calculi terminate if this data structure has a particular form, i.e. if it is *closed*. In this case a translation of the computed data structure to a demand expression, the so-called *standard representation*, is defined. The solutions of the analysis problem are obtained as a specific subset of the standard representation's concretization. An important property of the calculi, which is responsible for much of their power, is the use of global rules. In addition to a set of constraints available locally at a node the global rules have access to all the nodes along the path from the root. Such rules enable the calculi to detect loops and thus to solve constraints that without such rules would lead to non-termination.

We prove the soundness of ADE and soundness and completeness for CADE if it terminates.

Extensions and aspects of the implementation are dealt with in chapter 5. The extensions have all been prototypically implemented. The local rule's extensions in section 5.2 and the precision improvement for higher-order expressions have not been theoretically integrated to the same extent as the base calculi, in particular the extensions are not considered in the soundness and completeness proofs where only the base calculi are considered. From experience with the implementations as well as with the elaboration of the base calculi we conjecture that these extensions

could equally be integrated.
In chapter 6 we present different examples motivating some of the calculi's rules and hint toward their possible applications.

1.2 Related work

1.2.1 Abstract Interpretation

In the field of analyzing properties of programs for non-strict functional languages one of the most widespread methods may well be abstract interpretation [BHA85, AH87, Abr90a, Bur91, CC77, Myc81, Wad87]. Another method not quite as widespread is based on projections [HW87, Pat96, LPJ96].
Both methods have in common a definition based on the denotational semantics of the language analyzed.
For the denotational semantics domains are used which provide objects for the values of the language, e.g. $\mathbb{Z}$ for the syntactic constructs $\{\texttt{0}, \texttt{-1}, \texttt{1}, \texttt{-2}, \texttt{2}, \dots\}$. The mapping from syntactic constructs to domain elements is called *evaluation*.
If the language is of higher order, domains for continuous functions on domains are typically used as the co-domain of the evaluation function. Furthermore one defines an evaluation function, $[\![\cdot]\!]$, mapping language constructs into the domains. If one does this canonically, the standard interpretation is obtained. By the choice of evaluation function and domains, different non-standard interpretations are possible.
In this view a *property* of a program is a relation between subsets of the domain for the input values and those of the output values. It is important which domains are used, if there are e.g. only values for expressions with terminating computations resulting in some data value, we cannot associate a (sensible) semantic

value to expressions with non-terminating computations and thus the evaluation will only be a partial function. For many program analyses such domains are not an appropriate basis and typically *lifted* domains are used in the literature, i.e. domains that contain a value for expressions with non-terminating computations. Lifted domains are appropriate for many analyses, among them termination analysis, strictness analysis, absence analysis and demand analysis. Other analyses may need even richer domains, e.g. domains in which for every value the maximal number of normal-order reductions necessary to obtain it is attached to the value, the maximal number of heap accesses necessary in its computation or the maximal number of parallel threads into which the computation may be partitioned. With such domains and appropriate evaluation functions one obtains non-standard interpretations. We mention in passing that there are non-standard interpretations using domains smaller than those of the standard interpretation.

In order to deduce operational behavior from properties defined on a standard or non-standard interpretation the operational semantics needs to be related to the denotational semantics, i.e. the appropriate interpretation. This relation is *computational adequacy*. For the standard interpretation it amounts to:

· the operational semantics for a program results in a non-terminating reduction iff the standard interpretation maps the program to $\perp$, and

· the denotation of a program is not changed by reduction.

A great many properties of programs, required by optimizations and program transformations are undecidable.

The methods based on abstract interpretation abstract the domains, typically by using complete lattices in which a single value represents a subset of a domain. Abstract evaluation functions mapping language constructs to abstract values also need to be provided. The goal here is to be able to decide an abstraction of the property to be analyzed and obtain an approximation of the concrete property. The relation between the abstract interpretation and the standard or non-standard interpretation is given by a so called *Galois connection* [CC94]. The Galois connection consists of two functions, an abstraction and a concretization, where the abstraction maps concrete properties to abstract ones, i.e. to relations between abstract values and conversely the concretization maps abstract properties to concrete ones. Additionally, a Galois connection must be compatible with orders, i.e. the abstraction of a property must be smaller than an abstract value iff the concretization of the latter is contained in the property.

We want to relate our calculi, ADE and CADE, to abstract interpretation. The choice of the contextual order correlates to the choice of the domains for standard or non-standard interpretation.

The way the contextual order was chosen for our work, it is not feasible for the analysis of properties defined in terms of e.g. the number of reduction steps [San98a].

Our contextual order matches the standard interpretation.

As an example we consider strictness analysis, or context-analysis [FB94], based on Wadler's 4-point domain. Here the complete lattice $\perp^{\#} \leq \perp_{\infty} \leq \perp_{\in} \leq \top_{\in}$ is used. The concretization maps $\perp^{\#}$ to the expressions (of type list) without WHNF, $\perp_{\infty}$ to the set of expressions evaluating to infinite lists or to approximations thereof, $\perp_{\in}$ to the set of expressions evaluating to finite lists in which at least one element has no WHNF, and finally

$\top_\in$ to the set of expressions evaluating to finite lists in which all elements have a WHNF. The abstraction is defined accordingly and it is easy to see that we obtain a Galois connection between the standard interpretation and the abstract interpretation. The same sets may be represented as demands in ADE. For this representation some auxiliary demands are used. `Bot` stands for the same set as $\bot^\#$. Let $\texttt{Inf} = \langle \texttt{Bot}, \texttt{Top : Inf} \rangle$ and $\texttt{WInf} = \texttt{Top : Inf}$ then `WInf` and $\bot_\infty$ stand for the same sets. If $\texttt{Top}^+ = \langle \texttt{Fun}, \texttt{c}_{A,i}\ \texttt{Top} \ldots \texttt{Top}, \ldots \rangle$ stands for all expressions having a WHNF and $\texttt{Fin} = \langle \texttt{[]}, \texttt{Top : Fin} \rangle$ stands for all finite lists with arbitrary elements then $\texttt{WBotElem} = \langle \texttt{[]}, \texttt{Bot : Fin}, \texttt{Top}^+ : \texttt{WBotElem} \rangle$ stands for the same set as $\bot_\in$. Finally, $\top_\in$ corresponds to $\texttt{WTopElem} = \langle \texttt{[]}, \texttt{Top}^+ : \texttt{WTopElem} \rangle$. It is straightforward to see that (our) concretization of the demands really does result in the sets above.

Thus we can connect analyses of ADE with analyses using abstract interpretation. If e.g. the analysis of $\texttt{length}\ xs \in \texttt{Bot} \to_{\text{ADE}} \texttt{WInf}$, then for every value $s \in \gamma(\texttt{WInf})$ we know $\texttt{length}\ s \in \gamma(\texttt{Bot})$. Due to computational adequacy $\forall s \in \gamma(\texttt{WInf}) : [\![\texttt{length}\ s]\!] = \bot$ which abstracts to $\alpha(\texttt{length})\ \bot_\infty = \bot^\#$

Our calculi are obviously more precise than Wadler's 4-point domain. As an example consider the function $\texttt{second} \overset{\text{def}}{=} \lambda x.\texttt{head}\ (\texttt{tail}\ x)$. If an application of `second` is evaluated to WHNF the only evaluation degree for the argument expressible in the 4-point domain is evaluation to WHNF. Our calculi can automatically analyse `second`'s argument to need evaluation to WHNF, evaluation of the tail of that WHNF to WHNF, and evaluation of that tail's head to WHNF.

1.2.2 Set-based Analysis

The goal of set-based analysis is to constrain the set of values which may result from a part of a program. In order to achieve this the data-flow of the program is approximated. Depending on the kind of object language the program parts for which the value set is approximated may be the program variables or the program sub-expressions, etc.

Set-based analysis is not limited to functional languages, instead it was formulated for a simple imperative language e.g. in [JM79]. Another example of an analysis for a language with imperative features can be found in [FF95]. There full Scheme with assignment and first-class continuations is considered. In [HJ92, Mis84, GdW94] the approach is applied to logic programs. Still it seems the majority of research on set-based analysis is concerned with functional languages [AW93, Bis97, AWL94].

Set-based analysis works in two phases: the first, *specification*, computes set-constraints from the program text and the second, *solution*, computes a minimal model solving the constraints.

The set-constraints are inclusion relations between pairs of set-expressions, where the set-expressions are in turn constructed from some given operations on set-expressions (e.g. union, intersection, complement etc.), set-variables, for which the system of set-constraints is to be solved, and function symbols, typically uninterpreted. The choice of operations allowed for the set-expressions essentially influences the time complexity of the solution phase.

The set-variables represent a notion of *locality* in the program text. This notion will vary with the language analyzed and with the kind of analysis: for e.g. imperative languages a distinct set-variable might be chosen for any pair of program statement and program variable, as in [HJ94], to constrain the set of values of a program variable at a particular program position. For functional languages set-variables may e.g. be chosen for marked subexpressions as in [Bis97] or for heap-locations [FF95] if these are visible in the syntax of the language and these set-variables will then constrain the set of values which may appear in corresponding locations.

For the function symbols the constructors of the object language may be chosen or as in [FF95] the constructors of heap-values themselves may be used. These heap-values carry references to heap-locations, but since in the cited work heap-locations correspond one-to-one with set-variables no problems arise and a technically simple presentation is achieved.

Depending on the features to be analyzed different definitions of *value* are used which also affects the set-based semantics. As an example [Bis97] uses an additional symbol $\perp\!\!\!\perp$ for expressions, which do not contribute to the result of the program.

The development of a set-based analysis for a semantic feature starts with an appropriate environment based operational semantics for the feature. In general this is *not* the standard semantics of the language, but it can be as e.g. in [FF95]. For many set-based analyses a natural semantics, i.e. a big-step operational semantics, is used. Again this is not necessarily so, as [FF95] shows, by using a small-step operational semantics or as [AWL94, AW93] show by using a semantics of type inference.

The environment, which maps free variables to values of the semantics will in a next step be converted to a set-environment having the same domain as the environment, but mapping to sets of values. This entails the presentation of a set-based semantics for which the values resulting from the original semantics are

elements of the results of the set-based semantics. The set of values computed as the result of a program in the minimal safe set-environment is the set-based approximation of that program. The actual specification is then performed by a syntax-directed translation of the program text into a set of set-constraints which in turn must have a minimal model mapping the program to its set-based approximation.

There is a wealth of analyses which employ this framework with object languages ranging from functional via logic to object-oriented, analyzed features including among others types or absence of expressions from the computation and set-constraint languages ranging from very restricted languages that do not allow arbitrary intersection, union, complement etc. to unrestricted set-constraint languages that may include these operations, projection, function-spaces, etc. Accordingly, the running times of the analyses which are dominated by the specification phase vary from polynomial time to non-deterministic exponential time.

The essential step which is on the one hand responsible for making the analyses decidable and which on the other hand forces the inaccuracy is the translation into the language of set-constraints. A fundamental difference between ADE and the set-based analyses found in the literature is therefore that in the computation of a solution using set-based analyses no reduction is used, but in ADE this is well used.

ADE as SBA?

We will now set to work on the difference between ADE and SBA. Obviously, both are methods to formulate various program analyses. The result of ADE is an expression of the demand language, Λ_C, constraining the values possible for variables in the input.

This most closely corresponds to a system of set-constraints as it is produced by SBAs in the specification phase. Under these premises the demand language corresponds to the language of set-constraints and ADE corresponds to a specification phase. We do not provide anything which would correspond to a solution phase: an appropriate algorithm does not exist. This is a consequence of our result that even simple questions such as emptiness of a demand are undecidable. It follows that arbitrary demands cannot be encoded as systems of set-constraints, since these properties are decidable for set-constraints. We will not give a characterization of the subset of demands which can be so encoded in the present work. This is left for future work to investigate. A further aspect which we can compare is the way by which the specification is obtained. In SBAs a syntax-directed translation is used whereas ADE may use, among other things, reduction of Λ-expressions and in the loop rules has access to entire computation paths. Comparing ADE to SBA shows that while superficially ADE may be viewed as the specification phase of an SBA, looking a little closer reveals great differences opposing an integration of ADE into the SBA framework.

1.2.3 Other related work

Recently, Dirk Pape classified strictness analyses in his dissertation [Pap00]. According to this classification and that in e.g. [DW90] the calculi we present are backwards analyses. Furthermore, our calculi are closest to the analyses with an infinite analysis domain, although our demands, Λ_C, do not form a semantic domain.
The definition of *demands* in this dissertation is different from that used e.g. in [Tre94] or in [Pap00]. In [Pap00] the language for for-

mulating the demands (as Scott-closed subsets of the appropriate domain) is not limited whereas in [Tre94] demands are limited to be graph structures defined through the use of a demand environment and appropriate bindings. In contrast to [Pap00] we have a restricted language of demands, Λ_C, but in contrast to [Tre94] the complexity of our demand language equals that of Turing-machine computations (cf. section 3.4).
Nöcker [Nöc93] uses *abstract values* with a structural complexity similar to Tremblay's demands, but the latter represent only expressions with a structure of finite depth while the former can also be used to e.g. represent infinite lists.
The idea behind the global rules in this dissertation is based on [Nöc93] and on the experience from [Sch94]. There *reduction path analysis* is used detecting loops based on the abstract reduction history of an expression.

2 Language

In this chapter we define the language to be analyzed, Λ. Λ is a λ-calculus to which we add constructors and a case. While we do not require every expression to be typed in the sense of the typed λ-calculus, we do intend Λ to be used as a core language in the compilation of a higher level typed functional language. Consequently, the constructors will belong to types of the higher level language which are formed from type constructors. Keeping the type constructor attached to the constructors as well as to the case construct will allow us to rule out many expressions as ill-typed and will match Λ more closely to a higher level functional language.

2.1 Syntax

2.1.1 Types

The type system of Λ consists of a set $\mathcal{A}$ of type constructor names, $A_1, \ldots, A_n$, each of which stands for a set $\mathtt{c}_A$ of data constructor names, $\mathtt{c}_{A,1}, \ldots, \mathtt{c}_{A,|A|}$. These sets are disjoint for any two different types. The number of constructors for a type A is denoted $|A|$. Every constructor $\mathtt{c}$ has an arity $\alpha(\mathtt{c})$, and an index within $\mathtt{c}_A$, such that we can speak of the first, second , ..., $|A|^{\text{th}}$ constructor of A. The constructor with index j of A is denoted $\mathtt{c}_{A,j}$. For every type $A \in \mathcal{A}$ there is a case-constant $\mathtt{case}_A$

of arity $|A|+1$. Finally, we allow an infinite set of variables. We assume that all the permitted symbols are different. The only way to discriminate among the data constructors of a type constructor A from within a program is by means of $\mathtt{case}_A$.

Definition 2.1 (syntax of Λ). We define Λ to be the language generated by the grammar $G = (\{K, V, E, C\}, \{\langle\text{varid}\rangle\} \cup \{\mathtt{c}_{A,i} | A \in \mathcal{A}, 1 \le i \le |A|\}, P, E\}$ where P is the set of productions in 2.1.

constructor	$K \rightarrow$	$\mathtt{c}_{A,i}, A \in \mathcal{A}, 0 \le i \le \lvert A \rvert$
case	$C \rightarrow$	$\mathtt{case}_A, A \in \mathcal{A}$
variable	$V \rightarrow$	$\langle\text{varid}\rangle$
application	$A \rightarrow$	$(E\ E)$
abstraction	$L \rightarrow$	$(\lambda V.E)$
expression	$E \rightarrow$	$K \mid V \mid A \mid L \mid C$

Table 2.1: The set P of productions for Λ's grammar

Whenever there is no risk of confusion we will, for ease of notation, write e.g. $t \in E$ or $t \in \Lambda$ to express the fact that t is an expression instead of $t \in L(G)$ where $G = (\{K, V, E, C\}, \{\langle\text{varid}\rangle\} \cup \{\mathtt{c}_{A,i} | A \in \mathcal{A}, 1 \le i \le |A|\}, P, E\})$.

Definition 2.2 (position, level). Formally, a term t may be regarded as a total function, mapping elements of a finite set of sequences of the naturals, the *positions*, to terminals or strings of terminals produced by some non-terminal of the grammar above, i.e. the *subexpressions* or *terms*, where the domain is prefix-closed, finite, and $\forall i : s_1 \ldots s_n i$ in the domain of t implies that

$s_1 \dots s_n(i-1)$ is also in this domain. The length of a subexpression's position will also be called the *level* of the subexpression. At level 0, the subexpression is the entire term. The depth of a term is the maximal level of its subexpressions, i.e. the maximal length of a position within the term.

Notation 2.3. We use some notation and conventions throughout this work to facilitate comprehensibility. These are used where appropriate and will not need to be specifically mentioned.

1. Letters $e, f, g, r, s, \dots, w$ denote Λ-expressions.
2. Letters x, y, z denote variables.
3. We write $e_1 \dots e_n$ for $(\dots(e_1\ e_2)\dots e_n)$.
4. We use Barendregt's variable convention [Bar84].
5. $V(s)$ denotes the set of (all) variables in an expression s.
6. We write $\lambda x_1, \dots, x_n.e$ for $\lambda x_1. \dots \lambda x_n.e$.
7. $s^i\ t$ abbreviates $\underbrace{s\ (s \dots (s}_{i}\ \ t) \dots)$.

Definition 2.4. For words w, v of a language L over alphabet Σ we define the *structural* or *syntactic order* (or *substring relation*) to be

$$w \le v \iff \exists \alpha, \beta \in \Sigma^* : \alpha w \beta = v.$$

We state without proof that this is a partial order and a precongruence, and we call $\equiv \overset{\text{def}}{=} \le \cap \ge$ *syntactic equivalence*. Furthermore, $< \overset{\text{def}}{=} \le \cap \not\ge$.

Notation 2.5. For any relation $\rightarrow \subseteq A \times A$ we write $\xrightarrow{+}$ for the transitive, $\xrightarrow{*}$ for the reflexive transitive closure and $\xrightarrow{n}$ for the n-fold iteration of $\rightarrow$.

2.1.2 Substitutions

Definition 2.6. An occurrence of a variable is *free* if it is not in the scope of a λ binder for that variable. The set $\mathcal{FV}(t)$ of variables occurring free in an expression $t \in \Lambda$ is

$$\begin{aligned}
\mathcal{FV}(\mathtt{c}_{A,i}) &\overset{\text{def}}{=} \emptyset \\
\mathcal{FV}(\mathtt{case}_A) &\overset{\text{def}}{=} \emptyset \\
\mathcal{FV}(v) &\overset{\text{def}}{=} \{v\}, \text{ if } v \in V \\
\mathcal{FV}(E_1\ E_2) &\overset{\text{def}}{=} \mathcal{FV}(E_1) \cup \mathcal{FV}(E_2) \\
\mathcal{FV}(\lambda v.E) &\overset{\text{def}}{=} \mathcal{FV}(E) \setminus \{v\}
\end{aligned}$$

$\mathcal{FV}(\cdot)$ is canonically extended to sets of Λ expressions.

A term t without free variables, i.e. for which $\mathcal{FV}(t) = \emptyset$ holds, is called *closed*, otherwise t is *open*. We write Λ^0 for the subset of Λ consisting of all closed expressions.
The notion of *substitution* is highly important in our work and we define two equivalent ways of expressing it.

Definition 2.7. A *substitution* σ maps Λ into Λ by replacing finitely many free variables with Λ-expressions. We write $\sigma = \{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$ for variables $x_1, \dots, x_n$ and Λ-expressions $t_1, \dots, t_n$. The *domain* of σ is defined as $\text{dom}(\sigma) \overset{\text{def}}{=} \{x_i | x_i \ne t_i\}$ and $\text{cod}(\sigma) \overset{\text{def}}{=} \{t_i | x_i \ne t_i\}$ is called σ's *co-domain*. If every element of the co-domain is closed, σ is *ground*. The *application of substitution* σ to Λ-expression s is defined as follows:

$$\begin{aligned}
\sigma x_i &\overset{\text{def}}{=} t_i \\
\sigma y &\overset{\text{def}}{=} y \text{ if } y \notin \{x_1, \dots, x_n\}
\end{aligned}$$

$$\sigma(\lambda y.s') \overset{\text{def}}{=} \lambda y.\sigma' s', \text{ where } \sigma' = \{x \mapsto t | x \neq y\}$$
$$\sigma(s_1\ s_2) \overset{\text{def}}{=} \sigma s_1\ \sigma s_2$$
$$\sigma \mathtt{c} \overset{\text{def}}{=} \mathtt{c}$$
$$\sigma \mathtt{case}_A \overset{\text{def}}{=} \mathtt{case}_A$$

The substitution mapping every variable to itself is called *id*.

Definition 2.8. As an alternative notation we define:

$$s[{}^{t_1}/_{x_1}, \ldots, {}^{t_n}/_{x_n}] \overset{\text{def}}{=} \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}s.$$

Definition 2.9. Let ρ, σ be substitutions with $\rho = \{x_1 \mapsto r_1, \ldots, x_n \mapsto r_n\}$, then we define

$$\rho^\sigma \overset{\text{def}}{=} \{x_1 \mapsto \sigma r_1, \ldots, x_n \mapsto \sigma r_n\}.$$

Remark 2.10. Let ρ, σ be substitutions, then

1. $\forall x \in \text{dom}(\rho) : \rho^\sigma x = \sigma\rho x$ and
2. $\text{dom}(\sigma) \cap \mathcal{FV}(\text{cod}(\rho)) = \emptyset \implies \rho^\sigma = \rho$.

In the first case $\rho^\sigma x_i = \sigma r_i$ and $\sigma\rho x_i = \sigma r_i$, and in the second case for all $r_i : \sigma r_i = r_i$.

Definition 2.11. Let σ and ρ be substitutions and let Σ be a set of substitutions. Suppose $s \in \Lambda$ and $S \subseteq \Lambda$. We define

$$\sigma S \overset{\text{def}}{=} \{\sigma s | s \in S\}$$
$$\Sigma s \overset{\text{def}}{=} \{\sigma s | \sigma \in \Sigma\}$$
$$\Sigma S \overset{\text{def}}{=} \bigcup_{\sigma \in \Sigma} \sigma S$$
$$\Sigma \pi \overset{\text{def}}{=} \{\sigma \pi | \sigma \in \Sigma\}.$$

The *substitution lemma* states how the sequence of two substitutions can be reversed to obtain identical terms. One has to take into account that variables may occur free in the substitutes of the substitution applied first that are also in the domain of the substitution applied last. This lemma corresponds to [Bar84, 2.1.16 Substitution Lemma].

Lemma 2.12 (substitution lemma). *Let σ, ρ be substitutions with disjoint domains and let dom$(\rho) \cap \mathcal{FV}(t) = \emptyset$ for all $t \in$ cod(σ), then*

$$\forall r \in \Lambda : \sigma\rho r \equiv \rho^\sigma \sigma r.$$

Proof. The proof proceeds, similarly to that in [Bar84], by induction on the structure of r.

$r \equiv \mathtt{c}$: By definition 2.7 $\forall\sigma : \sigma\mathtt{c} \equiv \mathtt{c}$, so both sides are equal.

$r \equiv \mathtt{case}_A$: as above.

$r \equiv x \wedge x \in \text{dom}(\rho)$: $x \notin \text{dom}(\sigma)$ so $\rho^\sigma\sigma x \equiv \rho^\sigma x \equiv \sigma\rho x$.

$r \equiv y \wedge y \in \text{dom}(\sigma)$: $y \notin \text{dom}(\rho)$ thus both sides equal σy.

$r \equiv z \wedge z \notin \text{dom}(\rho) \cup \text{dom}(\sigma)$: Both sides equal z.

$r \equiv \lambda y.s$: By the variable convention, y is neither in the domain of the substitutions nor among the free variables of the substitutes.

$$\begin{aligned}
&\sigma\rho(\lambda y.s)\\
\equiv&(\text{variable convention } + \text{definition 2.7})\\
&\lambda y.\sigma\rho s\\
\equiv&(\text{induction hypothesis})\\
&\lambda y.\rho^\sigma\sigma s\\
\equiv&(\text{variable convention } + \text{definition 2.7})\\
&\rho^\sigma\sigma(\lambda y.s)
\end{aligned}$$

$r \equiv r_1\ r_2$: The statement is implied by the induction hypothesis.

□

Definition 2.13. Let θ be a function and $M \subseteq \text{dom}(\theta)$. The *restriction* of θ to M, written $\theta_{|M}$, is the function that maps elements of M to the same values as does θ and is undefined for values outside M. θ is an *extension* of σ if there is a set M for which $\theta_{|M} = \sigma$.

We abbreviate some combinators used in the rest of this work. These definitions are not super-combinators, in particular, they are not recursive, but serve merely as textual replacements.

Definition 2.14.

$$\begin{aligned}
\mathtt{K} &\overset{\text{def}}{=} \lambda x.\lambda y.x\\
\mathtt{K}_m &\overset{\text{def}}{=} \lambda x_0.\dots\lambda x_m.x_0\\
\mathtt{D} &\overset{\text{def}}{=} \lambda f.\lambda x.(f\ (x\ x))\\
\mathtt{Y} &\overset{\text{def}}{=} \lambda f.(\mathtt{D}\ f\ (\mathtt{D}\ f))
\end{aligned}$$

$$\begin{aligned}
\mathtt{A} &\overset{\text{def}}{=} \lambda z.\lambda x.(x\ (z\ z\ x))\\
\boldsymbol{\theta} &\overset{\text{def}}{=} \mathtt{A}\ \mathtt{A}\\
\mathtt{id} &\overset{\text{def}}{=} \lambda x.x\\
\mathtt{bot} &\overset{\text{def}}{=} \boldsymbol{\theta}\ \mathtt{id}\\
\mathtt{proj}_{i,j} &\overset{\text{def}}{=} \lambda x_1.\dots\lambda x_i.x_j\\
\mathtt{bot}_n &\overset{\text{def}}{=} \lambda x_1.\dots\lambda x_n.\mathtt{bot}\\
\mathtt{sel}_{A,i,j} &\overset{\text{def}}{=} \lambda x.(\mathtt{case}_A\ x\ \underbrace{\mathtt{bot}\dots\mathtt{bot}}_{j-1}\ \mathtt{proj}_{\alpha(\mathtt{c}_{A,i}),j}\ \underbrace{\mathtt{bot}\dots\mathtt{bot}}_{|A|-j})\\
\mathtt{det}_{A,i} &\overset{\text{def}}{=} \lambda x.(\mathtt{case}_A\ x\ \underbrace{\mathtt{bot}\dots\mathtt{bot}}_{i-1}\ \mathtt{c}_{A,i}\ \underbrace{\mathtt{bot}\dots\mathtt{bot}}_{|A|-i}).
\end{aligned}$$

2.2 Reduction

We take a contextual operational view on programming language semantics in this dissertation, i.e. the means by which we assign meaning to Λ-expressions involves reduction and programs with holes, so called contexts.
The operational semantics of Λ is given as a small step reduction semantics. We define an immediate reduction relation, $\rightarrow_B$, applicable only if the entire term has a specific form. Allowing immediate reduction at any node in an expression's syntax tree we obtain the notion of reduction, $\rightarrow$. Normal-order reduction, $\rightarrow_{\text{no}}$, is obtained by restriction of the contexts to reduction contexts.

2.2.1 Contexts

The notion of *contexts* presents a means to divide any expression into an "upper" part, the context, and a "lower" part, the subexpression which is inserted into the context at a specified position.

Definition 2.15 (context). A context is an expression with a single occurrence of an additional constant, $\cdot$, called *hole*. If t is an expression and p is a position in t, we may split t into a context $\mathbb{C}[\cdot]$ and a subexpression s such that $\mathbb{C}[\cdot]$ is defined for any position q of t's positions not having p as a prefix. On this domain $t(q) \equiv \mathbb{C}[\cdot](q)$ and $\mathbb{C}[\cdot](p) \equiv \cdot$ (also written $[\cdot]$) and $t_{|p} \equiv s$ so that $t \equiv \mathbb{C}[s]$.

Since we allow variables to be captured by $\mathbb{C}[\cdot]$, in general, it is not possible to allow arbitrary α-renaming for contexts.

Notation 2.16. For a context $\mathbb{C}[\cdot]$ we write $\mathbb{C}[\cdot] \in \Lambda$ or $\mathbb{C}[\cdot] \in \Lambda^0$ if for $s \in \Lambda$ or $s \in \Lambda^0 : \mathbb{C}[s] \in \Lambda$ or $\mathbb{C}[s] \in \Lambda^0$, respectively. We call $[\cdot]$ the *trivial context*.

Definition 2.17 (multi-context). A *multi-context*, $\mathbb{C}[\cdot_1, \ldots, \cdot_n]$ is an expression with occurrences of n distinct holes, each of which occurs exactly once. $\mathbb{C}[t_1, \ldots, t_n]$ is the expression which results from inserting the expressions $t_1, \ldots, t_n$ in place of the holes.

For multi-contexts the problem of variable capturing is complicated by the fact that every hole may appear in different scopes. Contexts which may duplicate holes are also used in the literature [San98b, Bar84], but for our presentations they are not necessary. We need to extend the definition of depth to contexts. If, for a previously unused variable a, the level of all subexpressions of $\mathbb{C}[a]$ differing from a is less than a's level, n, then the depth of $\mathbb{C}[\cdot]$ is $n-1$, otherwise the depth of $\mathbb{C}[\cdot]$ is the same as that of $\mathbb{C}[a]$. The operational semantics uses repeated β-reduction as in the λ-calculus, and convergence is defined with a stepwise evaluation relation using *reduction contexts* [San96, FFK87]. A reduction context is

a context with a single hole, which stands for the next expression to be evaluated.
The choice of reduction order forces the definition of reduction context. Here we want to achieve normal-order reduction and thus define reduction contexts as in definition 2.19.

Definition 2.18. A context $\mathbb{C}[\cdot]$ is below a context $\mathbb{D}[\cdot]$ in s if $\exists t : s \equiv \mathbb{D}[\mathbb{C}[t]]$.

Definition 2.19. A reduction context, $\mathbb{R}[\cdot]$, is inductively defined by the following grammar:

$$\begin{array}{rcl} \mathbb{R}[\cdot] & \rightarrow & [\cdot] \\ & | & \mathbb{R}[\cdot]\ e \\ & | & \texttt{case}_A\ \mathbb{R}[\cdot]\ e_1 \ldots e_{|A|} \end{array}$$

The alternative $e\,\mathbb{R}[\cdot]$ would mean that the argument may be evaluated before starting the evaluation of the function. While this might be desired, e.g. for optimization using strictness information, or for call-by-value, it deviates from normal-order reduction. In what follows, unless otherwise stated, the letters $\mathbb{Q}, \mathbb{R}, \mathbb{S}$ and $\mathbb{T}$ will denote reduction contexts.

Lemma 2.20. *If s can be written as $s \equiv \mathbb{R}[t] \equiv \mathbb{R}'[t']$ then either $\mathbb{R}[\cdot] \equiv \mathbb{R}'[\mathbb{S}[\cdot]]$ or $\mathbb{R}'[\cdot] \equiv \mathbb{R}[\mathbb{S}[\cdot]]$.*

Proof. In no production of the grammar for reduction contexts does $\mathbb{R}[\cdot]$ appear more than once. □

Definition 2.21. Let s be an expression. $\mathbb{R}[\cdot]$ is called *maximal reduction context for s*, if $s \equiv \mathbb{R}[t]$ and the only reduction context below $\mathbb{R}[\cdot]$ in s is the trivial reduction context.

Lemma 2.22. *$\mathbb{R}[\cdot]$ and $\mathbb{S}[\cdot]$ are reduction contexts, iff $\mathbb{R}[\mathbb{S}[\cdot]]$ is a reduction context.*

Proof. This follows directly from the definition 2.19 since holes, being trivial reduction contexts, appear only in positions where also a non-trivial reduction context could appear. □

Lemma 2.23. *Let σ be a substitution. $\sigma\mathbb{R}[\cdot]$ is a reduction context if $\mathbb{R}[\cdot]$ is a reduction context.*

Proof. Using induction on the structure of $\mathbb{R}[\cdot]$ it is easy to see that the substitution will not change this structure. □

The converse is not true in general as the following example shows.

Example 2.24. Let $\mathbb{C}[\cdot] \equiv x\ [\ ]\ \texttt{[]}\ (\lambda x.\lambda xs.x : xs)$ and let $\sigma = \{x \mapsto \texttt{case}_{\texttt{List}}\}$, then $\mathbb{C}[\cdot]$ is not a reduction context, but $\sigma\mathbb{C}[\cdot] \equiv \texttt{case}_{\texttt{List}}\ [\ ]\ \texttt{[]}\ (\lambda x.\lambda xs.x : xs)$ is a reduction context.

2.2.2 Reduction

Definition 2.25. The reduction relation, $\to$, is the smallest relation on expressions, satisfying the following conditions for all contexts, $\mathbb{C}[\cdot]$:

$$\mathbb{C}[s] \to \mathbb{C}[t], \text{if } s \to_B t \tag{2.1}$$

where

$$(\lambda x.t)\ e \to_B t[^e/_x] \tag{2.2}$$

$$\texttt{case}_A\ (\texttt{c}_{A,i}\ \vec{e})\ t_1 \dots t_{|A|} \to_B t_i\ e_1 \dots e_{\alpha(\texttt{c}_{A,i})} \tag{2.3}$$

Notation 2.26. It will be convenient to use vector notation $\vec{e}$ to stand for "all the $e_1, \dots, e_n$" if the intended meaning is clear from the context. So for (2.3) we may write

$\texttt{case}_A\ (\texttt{c}_{A,i}\ e_1 \dots e_{\alpha(\texttt{c}_{A,i})})\ t_1 \dots t_{|A|} \to_B t_i\ \vec{e}$ or even $\texttt{case}_A\ (\texttt{c}_{A,i}\ \vec{e})\ \vec{t} \to_B t_i\ \vec{e}$ and for a multi-context $\mathbb{C}[\cdot_1, \dots, \cdot_n]$ we may write $\mathbb{C}[\vec{e}]$ for $\mathbb{C}[e_1, \dots, e_n]$. We will even write $s[\overrightarrow{^t/_x}]$ for the substitution $s[^{t_1}/_{x_1}, \dots, ^{t_n}/_{x_n}]$ if it is evident from the context which $t_1, \dots, t_n$ are intended.

Definition 2.27 (redex, contractum). A subexpression s of an expression t is called *redex*, if it can be reduced using the $\to_B$-relation, i.e. if there is a context, $\mathbb{C}[\cdot]$, such that $t \equiv \mathbb{C}[s] \wedge s \to_B s'$. s' is called the *reduct* of s and $\mathbb{C}[s']$ is called the *contractum* of t.

Definition 2.28 ($\to_{\text{no}}$, normal-order reduction). The reduction is said to be in *normal order*, if condition (2.1) is satisfied for a reduction context, i.e. if $s \equiv \mathbb{R}[s'], t \equiv \mathbb{R}[t']$ and $s' \to_B t'$. In this case we write $\to_{\text{no}}$ instead of $\to$. (This will *not* be a maximal reduction context.)

Remark 2.29. The reduction context of a normal-order redex is the one directly above the maximal reduction context.

A redex which can be reduced in normal order is called the *normal-order redex.* We call an expression s a *potential* redex, if there is a substitution σ such that σs is a redex in σt.

Lemma 2.30. *Let s be a closed Λ-expression. If t is a Λ-expression for which $s \to t$, then t is closed.*

Proof. It suffices to observe that Λ-expressions s', t' with $s' \to_B t'$ satisfy $\mathcal{FV}(s') \supseteq \mathcal{FV}(t')$. For $s \equiv \mathbb{C}[s']$, $\mathbb{C}[\cdot]$ will bind all the free variables of s', so the statement of the lemma is implied.

1. If $s' \equiv (\lambda x.s'')\ e$, then $\mathcal{FV}(s') = \mathcal{FV}(s'') \setminus \{x\} \cup \mathcal{FV}(e)$ and $t' \equiv s''[^e/_x]$. By the variable convention x is not among the

free variables of e and thus not among the free variables of t', but the other free variables of s'' and e are, thus $\mathcal{FV}(s') = \mathcal{FV}(t')$.

2. If $s' \equiv \texttt{case}_A\ (\texttt{c}_{A,i}\ e_1 \dots e_{\alpha(\texttt{c}_{A,i})})\ t_1 \dots t_{|A|}$, then $\mathcal{FV}(s') = \bigcup_i \mathcal{FV}(t_i) \cup \bigcup_i \mathcal{FV}(e_i)$ and $\mathcal{FV}(t') = \mathcal{FV}(t_i) \cup \bigcup_i \mathcal{FV}(e_i)$, and so obviously $\mathcal{FV}(s') \supseteq \mathcal{FV}(t')$. □

Proposition 2.31. *The normal-order redex is uniquely defined.*

Proof. Let $r, r' \in \Lambda$ and $\mathbb{R}[\cdot], \mathbb{R}'[\cdot]$ be reduction contexts. Assume r has more than one normal-order redex. By lemma 2.20 it suffices to consider $r \equiv \mathbb{R}[s]$ where $s \equiv \mathbb{R}'[t]$. We can reduce

$$r \equiv \mathbb{R}[s] \rightarrow_{\text{no}} \mathbb{R}[s'] \equiv r' \text{ by } s \rightarrow_B s'$$
$$s \equiv \mathbb{R}'[t] \rightarrow_{\text{no}} \mathbb{R}'[t'] \equiv s' \text{ by } t \rightarrow_B t'$$

$\mathbb{R}[\cdot] \equiv \mathbb{R}[\mathbb{R}'[\cdot]]$, i.e. $\mathbb{R}'[\cdot] \equiv [\cdot]$ since $\mathbb{R}[\cdot]$ is the reduction context directly above the maximal reduction context and since there cannot be a $\rightarrow_B$-reduction in the hole of a maximal reduction context, because in every expression being $\rightarrow_B$-reducible there must be a non-trivial reduction context. □

Lemma 2.32. *Let $\mathbb{R}[\cdot]$ be a reduction context, $s, t \in \Lambda$ and $s \xrightarrow{*}_{no} t$, then*

$$\mathbb{R}[s] \xrightarrow{*}_{no} \mathbb{R}[t].$$

Proof. By induction on the length of the normal-order reduction $s \xrightarrow{*}_{\text{no}} t$ observing that the $\rightarrow_B$-reductions will not affect the reduction context. □

The contextual representation of Λ-expressions is highly expressive, but one needs to be careful not to be misled. Namely, one

could come to believe $\mathbb{C}[s] \xrightarrow{*}_{\text{no}} \mathbb{R}[s] \implies \mathbb{C}[t] \xrightarrow{*}_{\text{no}} \mathbb{R}[t]$, which is false.

Example 2.33. Let $f \in \Lambda$ be some expression reducing to an abstraction and let $g, h \in \Lambda$. Assume $f\ g \xrightarrow{*}_{\text{no}} \mathbb{R}[g]$ and that this is the first time the normal-order redex is within g. If this implied $f\ h \xrightarrow{*}_{\text{no}} \mathbb{R}[h]$ then in particular $f\ g \xrightarrow{*}_{\text{no}} \mathbb{S}[g\ v] \implies f\ h \xrightarrow{*}_{\text{no}} \mathbb{S}[h\ v]$. But this is false! A simple counter-example is $f \equiv \lambda x.(x\ x)$ for which we have $f\ g \xrightarrow{*}_{\text{no}} g\ g$ and $f\ h \xrightarrow{*}_{\text{no}} h\ h$, but in general $g \not\equiv h$.

2.2.3 Stability of reduction

The $\rightarrow_B$-reduction is stable with respect to arbitrary substitution of free variables. That is, if some expression can be $\rightarrow_B$-reduced and this expression has free variables, then it does not matter what is substituted for the free variables, the same reduction will be possible after substitution.
We generalize stability to other properties, i.e. we will speak of some property being stable with respect to some operation, if that property is not changed by the operation. The property to be considered now is $\rightarrow_B$-reducibility of some redex and the operation is substitution of free variables.

Lemma 2.34. *Let $s, t \in \Lambda$ with $\mathcal{FV}(s) \cup \mathcal{FV}(t) = \{x_1, \dots, x_n\}$. Iff $s \rightarrow_B t$ then*

$$\forall r_1, \dots, r_n \in \Lambda : s[^{r_1}/_{x_1}, \dots, ^{r_n}/_{x_n}] \rightarrow_B t[^{r_1}/_{x_1}, \dots, ^{r_n}/_{x_n}].$$

Proof.

$\implies$: We distinguish the two possibilities for the $\rightarrow_B$-reduction.

1. If $s \equiv (\lambda y.s')\ s''$ then as a consequence of the variable convention y is neither free in s nor in the s_i and $t \equiv s'[{}^{s''}/y]$.

$$\begin{aligned}
&(\lambda y.s'\ s'')[\overrightarrow{{}^{r}/x}]\\
\equiv&(\text{definition 2.7 } + \text{ variable convention})\\
&(\lambda y.s'[\overrightarrow{{}^{r}/x}])\ s''[\overrightarrow{{}^{r}/x}]\\
\to_B&\\
&s'[\overrightarrow{{}^{r}/x}][{}^{s''[\overrightarrow{{}^{r}/x}]}/y]\\
\equiv&(\text{variable convention} + \text{ lemma 2.12 with}\\
&\sigma = [\overrightarrow{{}^{r}/x}] \text{ and } \rho = \{y \mapsto s''\})\\
&s'[{}^{s''}/y][\overrightarrow{{}^{r}/x}].
\end{aligned}$$

2. If $s \equiv \texttt{case}_A\ (\texttt{c}_{A,i}\ e_1 \dots e_{\alpha(\texttt{c}_{A,i})})\ t_1 \dots t_{|A|}$ then $t \equiv t_i\ e_1 \dots e_{\alpha(\texttt{c}_{A,i})}$.

$$\begin{aligned}
&(\texttt{case}_A\ (\texttt{c}_{A,i}\ e_1 \dots e_{\alpha(\texttt{c}_{A,i})})\ t_1 \dots t_{|A|})[\overrightarrow{{}^{r}/x}]\\
\equiv&(\text{definition 2.7 iterated})\\
&\texttt{case}_A\ (\texttt{c}_{A,i}\ e_1[\overrightarrow{{}^{r}/x}] \dots e_{\alpha(\texttt{c}_{A,i})}[\overrightarrow{{}^{r}/x}])\ \overrightarrow{t[{}^{r}/x]}\\
\to_B&\\
&t_i[\overrightarrow{{}^{r}/x}]\ e_1[\overrightarrow{{}^{r}/x}] \dots e_{\alpha(\texttt{c}_{A,i})}[\overrightarrow{{}^{r}/x}]\\
\equiv&(\text{definition 2.7 right-to-left})\\
&(t_i\ e_1 \dots e_{\alpha(\texttt{c}_{A,i})})[\overrightarrow{{}^{r}/x}].
\end{aligned}$$

$\Longleftarrow$: We can substitute the free variables themselves to obtain this implication.

□

The $\to$-reduction and the $\to_{\text{no}}$-reduction inherit the stability with respect to arbitrary substitution from the $\to_B$-reduction.

Proposition 2.35. *Lemma 2.34 also holds for* $\to$ *and* $\to_{no}$.

Proof. We will argue for $\to_{\text{no}}$ only, for $\to$ the argument is analogous.

$\Longrightarrow$: Let $\mathbb{R}[\cdot]$ be a reduction context with $s \equiv \mathbb{R}[s']$ and $t \equiv \mathbb{R}[t']$ and $s' \to_B t'$. Due to lemma 2.34 we can apply the same reduction to an arbitrarily substituted s' and obtain the appropriately substituted t'. From lemma 2.23 we know that the substituted reduction context is also a reduction context and hence the statement holds.

$\Longleftarrow$: If $s[\overrightarrow{{}^{r}/x}] \to_{\text{no}} t[\overrightarrow{{}^{r}/x}]$ holds for any choice of $r_1, \dots, r_n \in \Lambda$, then in particular it will hold for $r_i \equiv x_i$. □

2.2.4 Weak head normal form

Definition 2.36. An expression is in *weak head normal form, WHNF*, if it is in one of the following forms:

- $\texttt{c}\ e_1 \dots e_n$ where $\texttt{c}$ is a constructor of arity n,
- $\lambda x.s$ or $\texttt{c}\ e_1 \dots e_n$ where $\texttt{c}$ is a constructor or a $\texttt{case}_A$ constant and the arity of $\texttt{c}$ exceeds n

A WHNF of the first kind is called *saturated constructor WHNF, SCWHNF,* and one of the second kind is called *function WHNF, FWHNF.*

WHNFs are the *values* in our work. This is in accordance with current practice in implementing lazy functional programming languages and e.g. with [Abr90b]. Intuitively, this amounts to regarding expressions which after a finite reduction sequence reduce to a WHNF as *convergent*. It may seem obvious to view all other expressions as *divergent*. Among these, though, there are some that reduce to a reduction context in which a variable is in the hole, e.g. $\mathbb{R}[x]$. These are not WHNFs, but some may reduce to WHNF for appropriate substitution of the variable and some may not reduce to WHNF no matter what is substituted. In a way, the $\mathbb{R}[x]$ and expressions which reduce to this form are "amorphous". We decide not to call expressions of the form $\mathbb{R}[x]$ divergent, instead we call only those expressions divergent that have no WHNF and that do not reduce to an expression of the form above. In the rare case where we will need a name for the kind of expressions having the form $\mathbb{R}[x]$, we will call them *suspended* expressions. We also speak of *non-convergent* and *non-divergent* expressions in case they are divergent or suspended or in case they are convergent or suspended, respectively.
An expression t has a WHNF, if an expression t_0 exists such that $t \xrightarrow{*} t_0$.

Our definition of WHNF seems to differ from that given in [PJ87]. Effectively, however, the difference is negligible. We distinguish suspended expressions from WHNFs only for the latter to be stable with respect to substitutions. In addition to (our) WHNFs and suspended expressions the definition of [PJ87] allows oversaturated constructor applications as WHNFs. But this effectively is no extension, since a well-typed program will not produce such expressions (cf. 2.4). In work based on the denotational standard semantics such expressions do not have a denotation since the interpretation is only defined for well-typed expressions. We take the stance that nothing relevant is lost by their exclusion.

Notation 2.37. Some notational conventions related to WHNFs and $s, t \in \Lambda$:

$s{\Downarrow}t$	if s has WHNF t
$s{\Downarrow}_S \mathtt{c}\ s_1 \dots s_{\alpha(\mathtt{c})}$,	if s has SCWHNF $\mathtt{c}\ s_1 \dots s_{\alpha(\mathtt{c})}$
$s{\Downarrow}_S$,	if $\exists t : s{\Downarrow}_S t$
$s{\Downarrow}_F t$,	if s has FWHNF t
$s{\Downarrow}_F$,	if $\exists t : s{\Downarrow}_F t$
$s{\not\Downarrow}$,	if s has no WHNF (s is non-convergent)
$s{\Downarrow}$,	if $\exists t : s{\Downarrow}t$ (s converges)
$s{\Uparrow}$,	if $s{\not\Downarrow} \wedge \forall x \in V, \mathbb{R}[\cdot] : s \not\xrightarrow{*} \mathbb{R}[x]$ (s diverges)

Corollary 2.38. *Closed expressions have closed WHNFs, if their reduction leads to WHNFs at all.*

Corollary 2.39. *$s \in \Lambda$ is a WHNF iff σs is a WHNF for any substitution σ.*

Proof. This is a corollary of proposition 2.35. □

2.2.5 Fixpoint combinators

Where convenient we use the fixpoint combinator $\boldsymbol{\theta}$ which goes back to Turing [Tur37]. The advantage of $\boldsymbol{\theta}$ over $\mathtt{Y}$ is syntactical: With $\boldsymbol{\theta}$ the original expression reappears as a sub-expression of the contractum after some reductions, with $\mathtt{Y}$ this is not the case.

Example 2.40.

$$\begin{aligned}\texttt{Y}\ s &\equiv (\lambda f.\texttt{D}\ f\ (\texttt{D}\ f))\ s\\ &\to_{\mathrm{no}} \texttt{D}\ s\ (\texttt{D}\ s)\\ &\equiv (\lambda f.\lambda x.f\ (x\ x))\ s\ (\texttt{D}\ s)\\ &\to_{\mathrm{no}} (\lambda x.s\ (x\ x))\ (\texttt{D}\ s)\\ &\to_{\mathrm{no}} s\ (\texttt{D}\ s\ (\texttt{D}\ s))\end{aligned}$$

compared to

$$\begin{aligned}\boldsymbol{\theta}\ s &\equiv (\lambda z.\lambda x.x\ (z\ z\ x))\ \texttt{A}\ s\\ &\to_{\mathrm{no}} (\lambda x.x\ (\texttt{A}\ \texttt{A}\ x))\ s\\ &\to_{\mathrm{no}} s\ (\texttt{A}\ \texttt{A}\ s)\\ &\equiv s\ (\boldsymbol{\theta}\ s)\end{aligned}$$

Anticipating section 2.5.1, we note that while the sub-expression $\texttt{D}\ s\ (\texttt{D}\ s)$ in $s\ (\texttt{D}\ s\ (\texttt{D}\ s))$ is contextually equivalent to $\texttt{Y}\ s$, this equivalence is not syntactical. On the other hand, starting reduction at $\boldsymbol{\theta}\ s$, we obviously arrive at an expression containing $\boldsymbol{\theta}\ s$ as a sub-expression and therefore prefer $\boldsymbol{\theta}$ as fixpoint combinator.

Corollary 2.41. *Let $\mathbb{R}[\cdot]$ be a reduction context.*

$$\mathbb{R}[\texttt{bot}] \xrightarrow{2}_{no} \mathbb{R}[\texttt{bot}].$$

Example 2.42. $\boldsymbol{\theta}\ \texttt{K}\ x \xrightarrow{*}_{\mathrm{no}} \texttt{K}\ (\boldsymbol{\theta}\ \texttt{K})\ x \to_{\mathrm{no}} \boldsymbol{\theta}\ \texttt{K}$.

2.3 Standardization and Invariance

Definition 2.43 (standardized reduction sequence). A reduction sequence is called *standardized*, iff all normal-order reductions occur before reductions not in normal order.

The following development establishes two essential results:

1. reduction sequences ending in a WHNF can be standardized, and
2. reduction of an expression does not change its "termination behavior", i.e. if an expression has a WHNF then so does its contractum and vice versa.

The two results replace the Church-Rosser property [Bar84] for our exposition since all we would use this property for is as a step on the way to establish results 1 and 2. In [SS99] the same is done for a slightly different language.
It will come as no surprise that our technique employs adapted notions from [Bar84], e.g. the 1-reduction.

Definition 2.44 ($\to_1$). The 1-reduction is defined as the following relation on Λ-expressions.

$$\begin{aligned}s &\to_1 s\\ s\ t &\to_1 s'\ t',\ \text{if } s \to_1 s' \text{ and } t \to_1 t'\\ \lambda x.s &\to_1 \lambda x.s',\ \text{if } s \to_1 s'\\ (\lambda x.s)\ t &\to_1 s'[^{t'}/x],\ \text{if } s \to_1 s' \text{ and } t \to_1 t'\\ \texttt{case}_A\ (\texttt{c}_{A,i}\ \vec{s})\ t_1 \dots t_{|A|} &\to_1 t'_i\ s'_1 \dots s'_{\alpha(\texttt{c}_{A,i})},\\ &\quad \text{if } t_i \to_1 t'_i \text{ and } \forall i : s_i \to_1 s'_i\end{aligned}$$

Remark 2.45. $\texttt{case}_A\ s\ t_1 \dots t_{|A|} \to_1 \texttt{case}_A\ s'\ t'_1 \dots t'_{|A|}$ can be obtained from definition 2.44 by inserting parentheses and applying induction on the term structure.

The intuition for the 1-reduction is a simultaneous reduction of multiple redexes visible in an expression. This is partly justified in lemma 2.46. In order to fully justify this intuition we would additionally have to show that any redex reduced was indeed already visible in the initial expression (cf. [Bar84]). This is omitted from our work since we make no use of it.

Lemma 2.46. *Let s be a Λ-expression. If $s \to_1 t$ then $s \overset{*}{\to} t$.*

Proof. The proof is by induction on the structure of s.
If $s \equiv \mathtt{c}$, $s \equiv \mathtt{case}_A$ or $s \equiv x$, then $t \equiv s$ must hold and so must the statement.
If $s \equiv s_1\ s_2$, then there are two possibilities for the 1-reduction.

1. $s_1\ s_2 \to_1 t_1\ t_2$ with $s_i \to_1 t_i$. The s_i are proper substructures of s, i.e. $s_i < s$, so by induction hypothesis $s_i \overset{*}{\to} t_i$. Then obviously $s \overset{*}{\to} t$.
2. $(\lambda x.s_1)\ s_2 \to_1 t_1[^{t_2}/x]$ with $s_i \to_1 t_i$. Again the $s_i < s$, so $s_i \overset{*}{\to} t_i$ and thus $(\lambda x.s_1)\ s_2 \overset{*}{\to} (\lambda x.t_1)\ t_2 \to t_1[^{t_2}/x]$.

If $s \equiv \lambda x.s_1$ then $s_1 < s$ and with $s_1 \overset{*}{\to} t_1$ we obtain $s \overset{*}{\to} t \equiv \lambda x.t_1$.
If $s \equiv \mathtt{case}_A\ s_0\ t_1 \ldots t_{|A|}$ we have two possibilities.

1. $\mathtt{case}_A\ s_0\ t_1 \ldots t_{|A|} \to_1 \mathtt{case}_A\ s_0'\ t_1' \ldots t_{|A|}'$ with $s_0 \to_1 s_0'$ and $\forall i : t_i \to_1 t_i'$. This case is covered under $s \equiv s_1\ s_2$.
2. $\mathtt{case}_A\ (\mathtt{c}_{A,i}\ s_1 \ldots s_{\alpha(\mathtt{c}_{A,i})})\ t_1 \ldots t_{|A|} \to_1 t_i'\ s_1' \ldots s_{\alpha(\mathtt{c}_{A,i})}'$. Induction gives $\mathtt{case}_A\ (\vec{s})\ \vec{t} \overset{*}{\to} \mathtt{case}_A\ (\mathtt{c}_{A,i}\ s_1' \ldots s_{\alpha(\mathtt{c}_{A,i})}')\ \vec{t'} \to t_i'\ \vec{s'}$. □

The 1-reduction is compatible with substitution in the sense that if an expression r accrues by substituting a variable occurring free in another expression s by an expression t then an expression r' can be obtained by applying the 1-reduction to s and t and

accordingly substituting. r' will then also be obtainable by the 1-reduction from r.

Lemma 2.47. *Let s, t be Λ-expressions with $s \to_1 s'$ and $t \to_1 t'$, then*

$$s[^t/x] \to_1 s'[^{t'}/x].$$

Proof. We show this using induction on the structure of s.

$s \equiv \mathtt{c}$, $s \equiv \mathtt{case}_A$ or $s \equiv y$ and $y \neq x$: $s[^t/x] \equiv s$ and $s'[^{t'}/x] \equiv s'$.

$s \equiv x$: $s[^t/x] \equiv t$ and $s'[^{t'}/x] \equiv t'$.

$s \equiv \lambda y.r$: Then $s' \equiv \lambda y.r'$ with $r \to_1 r'$ and r is a proper substructure of s, thus by the induction hypothesis $r[^t/x] \to_1 r'[^{t'}/x]$ and consequently $(\lambda y.r)[^t/x] \to_1 (\lambda y.r')[^{t'}/x]$.

$s \equiv r_1\ r_2$: The r_i are proper substructures of s and $s[^t/x] \equiv r_1[^t/x]\ r_2[^t/x]$. With the induction hypothesis $r_i[^t/x] \to_1 r_i'[^{t'}/x]$.

If $s' \equiv r_1'\ r_2'$ the statement holds, since $r_1'[^{t'}/x]\ r_2'[^{t'}/x] \equiv (r_1'\ r_2')[^{t'}/x]$.

If $s' \equiv r_3'[^{r_2'}/y]$ where $r_1 \equiv \lambda y.r_3$ and $r_3 \to_1 r_3'$, then the statement holds also, since $(r_3'[^{t'}/x])[^{r_2'[^{t'}/x]}/y] \equiv (r_3'[^{r_2'}/y])[^{t'}/x]$.

$s \equiv \mathtt{case}_A\ (\mathtt{c}_{A,i}\ s_1 \ldots s_{\alpha(\mathtt{c}_{A,i})})\ t_1 \ldots t_{|A|} \to_1 t_i'\ s_1' \ldots s_{\alpha(\mathtt{c}_{A,i})}'$:
Again we use the induction hypothesis to obtain $s_i[^t/x] \to_1 s_i'[^{t'}/x]$ and likewise for the t_i. The statement holds, since $t_i'[^{t'}/x]\ s_1'[^{t'}/x] \ldots s_{\alpha(\mathtt{c}_{A,i})}'[^{t'}/x] \equiv (t_i'\ s_1' \ldots s_{\alpha(\mathtt{c}_{A,i})}')[^{t'}/x]$. □

Corollary 2.48. *Let $t_1, \dots, t_n$ be Λ-expressions satisfying $\forall i : t_i \to_1 t'_i$ and let $\mathbb{C}[\cdot_1, \dots, \cdot_n]$ be a context. Then*

$$\mathbb{C}[t_1, \dots, t_n] \to_1 \mathbb{C}[t'_1, \dots, t'_n].$$

Proof. With $s \overset{\text{def}}{\equiv} \mathbb{C}[x_1, \dots, x_n]$ for fresh variables x_i we get $\mathbb{C}[t_1, \dots, t_n] \equiv s[{}^{t_1}/x_1] \dots [{}^{t_n}/x_n] \to_1 s[{}^{t'_1}/x_1] \dots [{}^{t'_n}/x_n] \equiv \mathbb{C}[t'_1, \dots, t'_n]$. □

If we apply a 1-reduction and a normal-order reduction to an expression, there are two possible situations which may result:

1. The 1-reduction includes the normal-order reduction. In this case another 1-reduction performing the remaining reduction will suffice to yield the initial 1-contractum from the normal-order contractum.
2. The 1-reduction does not include the normal-order reduction. In this case we can apply a normal-order reduction to the 1-contractum and a 1-reduction to the normal-order contractum to arrive at the same expression.

This situation is depicted in figure 2.1 and formalized in lemma 2.49 and corollary 2.50.

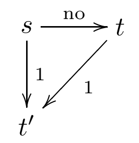
Lemma 2.49. *Let s be a Λ-expression. If $s \to_B t$ and $s \to_1 t'$, then either $t \to_1 t'$ or there is an expression u such that $t \to_1 u$ and $t' \to_B u$.*

Proof. We distinguish the two variants of the $\to_B$-reduction.

$s \equiv (\lambda x.s_1)\ s_2 \to_B s_1[{}^{s_2}/x]$: If $t' \equiv (\lambda x.s'_1)\ s'_2$ for $s_i \to_1 s'_i$ then $t' \to_B s'_1[{}^{s'_2}/x]$ and due to lemma 2.47 $t \to_1 s'_1[{}^{s'_2}/x]$.

If on the other hand $t' \equiv s'_1[{}^{s'_2}/x]$, then with the same lemma we conclude that $s_1[{}^{s_2}/x] \equiv t \to_1 t'$.

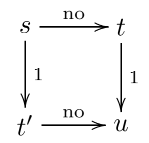
Figure 2.1: The two cases of corollary 2.50: The 1-reduction ...



$$\begin{tikzcd} s \arrow[r, "\text{no}"] \arrow[d, "1"] & t \arrow[dl, "1"] \\ t' & \end{tikzcd}$$

(a)

includes $\to_{\text{no}}$-reduction or



$$\begin{tikzcd} s \arrow[r, "\text{no}"] \arrow[d, "1"] & t \arrow[d, "1"] \\ t' \arrow[r, "\text{no}"] & u \end{tikzcd}$$

(b)

does not include $\to_{\text{no}}$-reduction

$s \equiv \texttt{case}_A\ (\texttt{c}_{A,i}\ s_1 \dots s_{\alpha(\texttt{c}_{A,i})})\ r_1 \dots r_{|A|} \to_B r_i\ s_1 \dots s_{\alpha(\texttt{c}_{A,i})}$:
If $t' \equiv \texttt{case}_A\ (\texttt{c}_{A,i}\ s'_1 \dots s'_{\alpha(\texttt{c}_{A,i})})\ r'_1 \dots r'_{\alpha(\texttt{c}_{A,i})}$, then $t' \to_B r'_i\ s'_1 \dots s'_{\alpha(\texttt{c}_{A,i})}$ and $t \to_1 r'_i\ s'_1 \dots s'_{\alpha(\texttt{c}_{A,i})}$ holds.

If on the other hand $t' \equiv r'_i\ s'_1 \dots s'_{\alpha(\texttt{c}_{A,i})}$ then $t \to_1 t'$ is already satisfied. □

Corollary 2.50. *Let s be a Λ-expression. If $s \to_{no} t$ and $s \to_1 t'$, then either $t \to_1 t'$ or there is an expression u such that $t \to_1 u$ and $t' \to_{no} u$.*

Proof. Lemma 2.49 and corollary 2.48 applied for appropriate reduction contexts. □

Theorem 2.51. *Let s be a Λ-expression. If $s \xrightarrow{*}_{no} t$ and t is a WHNF and $s \to s'$ then s' has a WHNF as well and it takes at most as many $\to_{no}$-reductions from s' to WHNF as it takes from s to t.*

Proof. It is easily seen that a reduction step can be simulated with the 1-reduction and thus $s \to_1 t'$. We use induction on the length k of the reduction path to t to prove the statement $s \xrightarrow{k}_{\text{no}} t$ and $s \to_1 s' \implies \exists t' : t \to_1 t'$ and $s' \xrightarrow{k'}_{\text{no}} t'$ with $k' \leq k$.

If $k = 0$ then $s \equiv t \equiv t'$ and s is in WHNF.

If $k > 0$ then we can apply corollary 2.50 with $s \rightarrow_{\text{no}} s_1$ and $s \rightarrow_1 s'$ and either $s_1 \rightarrow_1 s'$ or there is an s'_1 with $s_1 \rightarrow_1 s'_1$ and $s' \rightarrow_{\text{no}} s'_1$. In both cases the induction hypothesis can be applied for $s_1 \overset{k-1}{\rightarrow}_{\text{no}} t$ and $s_1 \rightarrow_1 s'_1$ (s'_1 may equal s') to obtain the statement. The procedure corresponds to appending commuting diagrams. See figure 2.3. $\square$

$$\begin{tikzcd}
s \arrow[r, "\rightarrow_{\text{no}}"] \arrow[d, "1"] & s_1 \arrow[r, "\rightarrow_{\text{no}}"] \arrow[d, "1"] & \cdots \arrow[r, "\rightarrow_{\text{no}}"] & t \arrow[d, "1"] \\
s' \arrow[r, "\rightarrow_{\text{no}}\cup\equiv"] & s'_1 \arrow[r, "\rightarrow_{\text{no}}\cup\equiv"] & \cdots \arrow[r, "\rightarrow_{\text{no}}\cup\equiv"] & t'
\end{tikzcd}$$

Figure 2.2: Appending commuting reduction diagrams

In the proof of theorem 2.51 an arbitrary $\rightarrow$-reduction at the beginning of a reduction sequence is converted to a 1-reduction following a normal-order reduction sequence. This 1-reduction however will in general reduce some redexes among which may be some normal-order redexes, as we saw in lemma 2.46. In order to observe this case more closely we introduce the notion of *involved normal-order reduction*, a 1-reduction, which also reduces a normal-order redex.

Definition 2.52. Let s be a Λ-expression. The reduction $s \rightarrow_1 t$ *involves normal-order reduction*, iff it is either of the form

$$\mathbb{R}[(\lambda x.s_1)\ s_2] \rightarrow_1 \mathbb{R}'[s'_1[^{s'_2}/x]],$$
$$\text{for } s_i \rightarrow_1 s'_i \wedge \mathbb{R}[\cdot] \rightarrow_1 \mathbb{R}'[\cdot]$$

or of the form

$$\mathbb{R}[\texttt{case}_A\ (\texttt{c}_{A,i}\ \vec{s})\ \vec{t}] \rightarrow_1 \mathbb{R}'[t'_i\ \vec{s'}],$$
$$\text{for } s_i \rightarrow_1 s'_i, t_i \rightarrow_1 t' \wedge \mathbb{R}[\cdot] \rightarrow_1 \mathbb{R}'[\cdot]$$

A 1-reduction is said to be *internal*, iff it does not involve normal-order reduction.

In the inductions above we used the structure of an expression to induce over. In lemma 2.54 this would not work out so easily since for e.g. $s \equiv s_1\ s_2$ with $s_i \overset{*}{\rightarrow}_{\text{no}} u_i \rightarrow_1 t_i$ it is not obvious how to combine the reduction sequences such that all normal-order reduction occurs before all internal reduction. The measure, ϕ, for 1-reductions defined below seems to be more appropriate for lemma 2.54 and will allow for a concise proof. Intuitively, $\phi(s,t)$ counts the number of $\rightarrow_{\text{no}}$- and $\rightarrow$-reduction steps necessary to achieve the same reduction as $s \rightarrow_1 t$ if $\rightarrow_{\text{no}}$-reductions are applied first.

Definition 2.53. Let $p \rightarrow_1 q$ be a 1-reduction.

$$\phi(p,q) \overset{\text{def}}{\equiv} \begin{cases} \phi(s,s') + \phi(t,t') & \text{if } p \equiv s\ t \wedge q \equiv s'\ t' \\ \phi(s,s') & \text{if } p \equiv \lambda x.s \wedge q \equiv \lambda x.s' \\ 1 + \phi(s,s') & \text{if } p \equiv (\lambda x.s)\ t \wedge q \equiv s'[^{t'}/x] \\ \quad + \phi(t,t') * \#_x s' & \text{where } \#_x s' \text{ is the number of free} \\ & \text{occurrences of } x \text{ in } s' \\ 1 + \phi(t_i, t'_i) & \text{if } q \equiv t'_i\ s'_1 \ldots s'_{\alpha(\texttt{c}_{A,i})} \\ \quad + \sum_j \phi(s_j, s'_j) & \wedge p \equiv \texttt{case}_A\ (\texttt{c}_{A,i}\ s_1 \ldots s_{\alpha(\texttt{c}_{A,i})})\ \vec{t} \\ 0 & \text{otherwise.} \end{cases}$$

With the following lemma we show that the normal-order reduction involved can be extracted from a 1-reduction.

Lemma 2.54. *Let s be a Λ-expression. If $s \to_1 t$ then we can find a Λ-expression u, for which $s \xrightarrow{*}_{no} u \to_1 t$ such that $u \to_1 t$ is internal.*

Proof. If the original 1-reduction is internal then this trivially holds.
Otherwise we consult definition 2.44 and definition 2.52 to see that $s \to_{\text{no}} u \to_1 t$ holds for an original 1-reduction involving normal-order reduction. We can iterate this process for the 1-reduction$u \to_1 t$. So we obtain an induction on $\phi(s,t)$.

- If $\phi(s,t) = 0$ the statement trivially holds.
- If $\phi(s,t) > 0$ and $s \to_1 t$ involves normal-order reduction then there are two cases.
  1. $s \equiv \mathbb{R}[(\lambda x.s_1)\ s_2] \to_{\text{no}} \mathbb{R}[s_1[^{s_2}/x]] \equiv u \to_1 \mathbb{R}'[s_1'[^{s_2'}/x]] \equiv t$. $\phi(u,t) < \phi(s,t)$ and from the induction hypothesis we obtain $u \xrightarrow{*}_{\text{no}} u' \to_1 t$ where $u' \to_1 t$ is internal.
  2. $s \equiv \mathbb{R}[\texttt{case}_A\ (\texttt{c}_{A,i}\ s_1 \dots s_{\alpha(\texttt{c}_{A,i})})\ t_1 \dots t_{|A|}] \to_{\text{no}} \mathbb{R}[t_i\ s_1 \dots s_{\alpha(\texttt{c}_{A,i})}] \equiv u \to_1 \mathbb{R}[t_i'\ s_1' \dots s'_{\alpha(\texttt{c}_{A,i})}] \equiv t$. Again $\phi(u,t) < \phi(s,t)$.

Summarizing, we can extract all the normal-order reduction steps involved in a 1-reduction and apply them first. □

Lemma 2.54 allows to split a 1-reduction into normal-order reductions followed by an internal 1-reduction. Splitting two consecutive 1-reductions in this way may give rise to a situation where we find normal-order reductions followed by an internal 1-reduction followed by normal-order reductions and again by an internal

1-reduction. To achieve standardization we will need to shift all of the second block of normal-order reductions to the left. Lemma 2.55 shows that this is indeed possible.

Lemma 2.55. *Let s be a Λ-expression. If $s \to_1 t \to_{no} u$ then the normal-order reduction can be shifted to the front of the reduction sequence, i.e. we can reduce $s \xrightarrow{+}_{no} t' \to_1 u$ where $t' \to_1 u$ is internal.*

Proof. We start by splitting $s \to_1 t \to_{\text{no}} u$ into $s \xrightarrow{*}_{\text{no}} r \to_1 t \to_{\text{no}} u$ where $r \to_1 t$ is internal (lemma 2.54). Since the 1-reduction is internal, we must consider only two cases:

1. $r \equiv \mathbb{R}[(\lambda x.r_1)\ r_2] \to_1 \mathbb{R}'[(\lambda x.r_1')\ r_2'] \equiv t \to_{\text{no}} \mathbb{R}'[r_1'[^{r_2'}/x]] \equiv u$, where $r_i \to_1 r_i'$ and $\mathbb{R}[\cdot] \to_1 \mathbb{R}'[\cdot]$. Now we can move the normal-order reduction to the front to obtain $r \equiv \mathbb{R}[(\lambda x.r_1)\ r_2] \to_{\text{no}} \mathbb{R}[r_1[^{r_2}/x]] \equiv t' \to_1 \mathbb{R}'[r_1'[^{r_2'}/x]] \equiv u$.
2. $r \equiv \mathbb{R}[\texttt{case}_A\ (\texttt{c}_{A,i}\ s_1 \dots s_{\alpha(\texttt{c}_{A,i})})\ t_1 \dots t_{|A|}] \to_1 \mathbb{R}'[\texttt{case}_A\ (\texttt{c}_{A,i}\ s_1' \dots s'_{\alpha(\texttt{c}_{A,i})})\ t_1' \dots t'_{|A|}] \equiv t \to_{\text{no}} \mathbb{R}'[t_i'\ s_1' \dots s'_{\alpha(\texttt{c}_{A,i})}] \equiv u$, where $t_i \to_1 t_i'$ and $\forall i : s_i \to_1 s_i'$ and $\mathbb{R}[\cdot] \to_1 \mathbb{R}'[\cdot]$. Again we can move the normal-order reduction to the front of the sequence to obtain $r \equiv \mathbb{R}[\texttt{case}_A\ (\texttt{c}_{A,i}\ s_1 \dots s_{\alpha(\texttt{c}_{A,i})})\ t_1 \dots t_{|A|}] \to_{\text{no}} \mathbb{R}[t_i\ s_1 \dots s_{\alpha(\texttt{c}_{A,i})}] \equiv t' \to_1 \mathbb{R}'[t_i'\ s_1' \dots s'_{\alpha(\texttt{c}_{A,i})}] \equiv u$.

In summary, for both cases, we have $s \xrightarrow{*}_{\text{no}} r \to_{\text{no}} t' \to_1 u$ where the 1-reduction is internal. □

Lemma 2.56. *Let s be a Λ-expression. A sequence $s \to_1 s_1 \to_1 s_2 \dots s_n$ with a WHNF s_n can be converted to a reduction sequence applying only normal-order reductions to arrive at a WHNF, i.e. $s \xrightarrow{*}_{no} s' \xrightarrow{*}_1 s_n$ for a WHNF s'.*

Proof. Without loss of generality n is the smallest index for which s_n is a WHNF. It follows that the reduction $s_{n-1} \to_1 s_n$ involves normal-order reduction and we split it with lemma 2.54 into $s_{n-1} \xrightarrow{k}_{\text{no}} s'_{n-1} \to_1 s_n$ where $s'_{n-1} \to_1 s_n$ is internal and $k \geq 0$. Then s'_{n-1} is a WHNF. Next we apply lemma 2.55 k times to shift the k normal-order reductions to the left over the 1-reduction $s_{n-2} \to_1 s_{n-1}$, i.e. to convert the sequence $s \ldots s_{n-2} \to_1 s_{n-1} \xrightarrow{k}_{\text{no}} s'_{n-1} \to_1 s_n$ into $s \ldots s_{n-2} \xrightarrow{k'}_{\text{no}} s'_{n-2} \to_1 s'_{n-1} \to_1 s_n$ in which the trailing 1-reductions are internal. It may be that $k' > k$, but the number of 1-reductions left of the k' normal-order reductions decreases with every one of these steps. Obviously this measure is well-founded. Furthermore we note, that after the ith step s'_{n-i-1} must be a WHNF, since all trailing 1-reductions are internal. So after finitely many of these steps we obtain the desired reduction sequence $s \xrightarrow{*}_{\text{no}} s' \xrightarrow{*}_1 s_n$ for a WHNF s'. □

Theorem 2.57 (Standardization). *Let s, t be Λ-expressions. If $s \xrightarrow{*} t$ for a WHNF t, then $\exists t_W \in \Lambda : t_W$ is a WHNF and $s \xrightarrow{*}_{no} t_W \xrightarrow{*} t$.*

Proof. We can simulate every $\to$-reduction with a 1-reduction. From lemma 2.56 we obtain a WHNF t_W satisfying $s \xrightarrow{*}_{\text{no}} t_W \xrightarrow{*}_1 t$. Then with lemma 2.46 we see $s \xrightarrow{*}_{\text{no}} t_W \xrightarrow{*} t$. □

Corollary 2.58. *Let $s \in \Lambda$. If there is no WHNF $t \in \Lambda$ such that $s \xrightarrow{*}_{no} t$, then s does not have a WHNF.*

Theorem 2.59 (Invariance of termination). *Let $s, t \in \Lambda$ with $s \to t$, then $s\Downarrow \iff t\Downarrow$.*

Proof.

$\Longrightarrow$: With the standardization theorem we find a WHNF s' reachable from s with a normal-order reduction sequence $s \xrightarrow{*}_{\text{no}} s'$. Now we can apply theorem 2.51.

$\Longleftarrow$: A reduction to WHNF of t implies a reduction to WHNF of s since $s \to t$. □

If we reconsider the proofs of lemmata and theorems in this section we recognize that the property of WHNFs that was actually used in those proofs is that WHNFs need normal-order reduction to arise from non-WHNFs and that once WHNF is reached further reduction does not change this property. We will now present this observation more formally.

Definition 2.60. A property F of Λ-expressions *needs normal-order reduction* if $\neg F(s) \wedge F(t) \wedge s \to_1 t$ implies that the $\to_1$-reduction involves normal order.

Definition 2.61. A property F is *stable* with respect to a relation $\mathcal{R}$ if $s\mathcal{R}t \wedge F(s) \implies F(t)$.

The only statements using WHNFs are in theorem 2.51, in lemma 2.56 and in theorems 2.57 and 2.59. If we replace the property of being a WHNF in these with any property satisfying the criteria in definitions 2.60 and 2.61 then one can easily verify that the proofs remain valid.
One property of Λ-expressions which satisfies these criteria is, of course, being a WHNF.

Lemma 2.62. *Let $s, t \in \Lambda$ and let $s \to_1 t \wedge s$ is no WHNF but t is a WHNF, then the $\to_1$-reduction involves normal-order reduction.*

Proof. The proof is by induction on the structure of s.

$s \equiv \mathtt{c}, s \equiv \mathtt{case}_A$ or $s \equiv x$: $s = t$ must hold, but then s is a WHNF iff t is a WHNF.

$s \equiv \lambda x.s'$: s is a WHNF.

$s \equiv s_1\ s_2$:

$t \equiv t_1\ t_2 \wedge s_i \rightarrow_1 t_i$: Since $t_1\ t_2$ is a WHNF, t_1 is an FWHNF and thus either $t_1 \equiv \mathtt{c}_{A,i}\ r_1 \ldots r_n$ where $n < \alpha(\mathtt{c}_{A,i})$ or $t \equiv \mathtt{case}_A\ r_0\ r_1 \ldots r_n$ where $n < |A|-1$. Since $s_1\ s_2$ is not a WHNF neither is s_1 and by the induction hypothesis $s_1 \rightarrow_1 t_1$ involves normal-order reduction. $\mathbb{R}[\cdot] \overset{\text{def}}{=} [\cdot]\ s_2$ is a reduction context and thus the normal-order reduction involved in $s_1 \rightarrow_1 t_1$ is also a normal-order reduction involved in $s \rightarrow_1 t$.

$(\lambda x.s_1')\ s_2 \rightarrow_1 t_1[{}^{t_2}/x] \wedge s_1' \rightarrow_1 t_1 \wedge s_2 \rightarrow_1 t_2$: Involves normal-order reduction by definition.

$s \equiv \mathtt{case}_A\ s_0\ t_1 \ldots t_{|A|}$: $\mathtt{case}_A\ (\mathtt{c}_{A,i}\ s_1 \ldots s_{\alpha(\mathtt{c}_{A,i})})\ t_1 \ldots t_{|A|} \rightarrow_1 t_i\ s_1 \ldots s_{\alpha(\mathtt{c}_{A,i})}$ involves a normal-order reduction by definition. The other cases with $\mathtt{case}_A$ are already covered above. □

Lemma 2.63. *Let $s, t \in \Lambda$ and let $s \rightarrow_1 t \wedge s$ is a WHNF, then t is a WHNF.*

Proof. We distinguish two cases.

$s \equiv \mathtt{c}_{A,i}\ s_1 \ldots s_n \wedge n \leq \alpha(\mathtt{c}_{A,i})$: Then $t \equiv \mathtt{c}_{A,i}\ t_1 \ldots t_n$ and $s_j \rightarrow_1 t_j$ thus t is a WHNF.

$s \equiv \mathtt{case}_A\ s_0\ s_1 \ldots s_n \wedge n \leq |A|$: Then $t \equiv \mathtt{case}_A\ t_0\ t_1 \ldots t_n$ and $s_j \rightarrow_1 t_j$ thus t is a WHNF. □

Lemma 2.64. *Further properties of a Λ-expression s satisfying the criterion in definition 2.60 are ($\mathbb{R}[\cdot]$ ranges over reduction contexts):*

1. *s has the form $\mathbb{R}[x]$ for a variable x,*
2. *s has the form $\mathbb{R}[\mathtt{c}_{A,i}\ t_1 \ldots t_n]$ where $n > \alpha(\mathtt{c}_{A,i})$,*
3. *s has the form $\mathbb{R}[\mathtt{case}_A\ t\ \ldots]$ where t is an FWHNF and*
4. *s has the form $\mathbb{R}[\mathtt{case}_A\ (\mathtt{c}_{A',i} \ldots)\ \ldots]$ where $A \neq A'$.*

Proof. Along the lines of the proof of lemma 2.62. □

Lemma 2.65. *The properties in lemma 2.64 are stable with respect to $\rightarrow_1$.*

2.4 Type system

Two main motivations steer our decisions with respect to type systems:

1. We want the type systems compatible with our calculi to be as diverse as possible, and
2. We want to focus on the rules essential for our analysis without complicating issues with type inference rules.

Together, these two points will hopefully ease implementation of the calculi for different functional languages, e.g. Haskell [PHA+99] or Cayenne [Aug98], and make the exposition clearer. In our effort to abstract from the concrete type system used we present criteria which a specific type system will have to meet if the calculi are to be used for corresponding languages. Furthermore the type system should be "rich" enough, i.e. there should be enough well-typed expressions. For example, it does not seem sensible to assume a type system in which no expression is well-typed: no expression would pass type check and being strongly typed the language would be utterly useless. We will require the type systems for the language analyzed to be at least as strong as the Damas-Hindley-Milner type system, i.e. for the set of well-typed expressions in the Damas-Hindley-Milner type system, WT^{DHM}, and the set of well-typed Λ-expressions, WT^{Λ}, we want $WT^{\mathrm{DHM}} \subseteq WT^{\Lambda}$. ADE and CADE are presented for a very general type system with WT as the set of its well-typed expressions. They may then be used for arbitrary type systems WT^{Λ}, satisfying $WT^{\mathrm{DHM}} \subseteq WT^{\Lambda} \subseteq WT$.

2.4.1 Well-typed expressions

First, we define expressions which are *directly ill-typed*. These are expressions without a normal-order reduction, but which are not in WHNF. For example $\mathtt{c}_{A,i}\ s_1 \dots s_n$, where $n > \alpha(\mathtt{c}_{A,i})$, or $\mathtt{case}_A\ t\ s_1 \dots s_{m_A}$, and t is a WHNF, but it does not permit the top level $\mathtt{case}_A$-reduction. This could e.g. be the case because t is an FWHNF. Next we define *ill-typed* expressions as those for which a normal-order reduction sequence leads to a reduction context with a directly ill-typed expression in the hole. Subsequently, we define the *well-typed* expressions as the complement of the ill-typed ones. All the ill-typed expressions will be treated as diverging.

Definition 2.66 (DIT, directly ill-typed).

$$\begin{aligned} DIT \overset{\text{def}}{=} \{t|&(t \equiv \mathtt{case}_A\ (\mathtt{c}_{A'}\ t_1 \dots t_{\alpha(\mathtt{c}_{A'})})\ e_1 \dots e_{|A|} \wedge A \neq A')\\ &\vee(t \equiv \mathtt{case}_A\ e\ e_1 \dots e_{|A|} \wedge e \text{ is a FWHNF})\\ &\vee(t \equiv \mathtt{c}\ t_1 \dots t_n \wedge n > \alpha(\mathtt{c}) \wedge \mathtt{c} \in K)\}.\end{aligned}$$

Ill-typed expressions are those for which the normal-order reduction sequence would have to reduce an expression in DIT.

Definition 2.67. Let s, t be expressions. We define

$$\begin{aligned} s \text{ crashes to } t \text{ iff } \exists r \in DIT, \mathbb{R}[\cdot] \in \Lambda : s \xrightarrow{*}_{\text{no}} t \wedge t \equiv \mathbb{R}[r]\\ s \text{ crashes iff } \exists t \in \Lambda : s \text{ crashes to } t\end{aligned}$$

An expression s which crashes is also called *ill-typed* and we write IT for the set of all ill-typed expressions. If s crashes to t, t is called an *ill-typed HNF, ITHNF*.

Remark 2.68. An expression remains directly ill-typed under arbitrary substitution of free variables. Consequently, an expression remains ill-typed under arbitrary substitution of free variables since the normal-order reduction sequence is stable with respect to substitution.

Lemma 2.69. *If s crashes its ITHNF is uniquely defined.*

Proof. This follows from the uniqueness of the normal-order redex. □

Remark 2.70. An expression $s \in DIT$ cannot be a redex, since there is no case of the $\rightarrow_B$ -reduction, which would be applicable.

Lemma 2.71. *Let $\mathbb{R}[\cdot]$ be a reduction context and let x be a variable, then*

$$\exists \mathbb{C}[\cdot], s \in \Lambda : \mathbb{C}[\mathbb{R}[x]] \in IT \wedge \mathbb{C}[s] \notin IT.$$

Proof. We distinguish the following cases for $\mathbb{R}[x]$.

$\mathbb{R}[\cdot] \equiv [\cdot]$: Let $\mathbb{C}[\cdot] \overset{\text{def}}{=} (\lambda x.[\cdot])\ (\mathtt{c}\ \mathtt{c})$ for some 0-ary constructor $\mathtt{c}$. $\mathbb{C}[\mathbb{R}[x]] \equiv (\lambda x.x)\ (\mathtt{c}\ \mathtt{c}) \rightarrow_{\text{no}} (\mathtt{c}\ \mathtt{c}) \in DIT$.

$\mathbb{R}[\cdot] \equiv \mathbb{R}'[\mathtt{case}_A\ [\cdot]\ e_1 \dots e_{|A|}]$: We can define $\mathbb{C}[\cdot] \overset{\text{def}}{=} (\lambda x.[\cdot])\ (\lambda y.y)$. Then we see $\mathbb{C}[\mathbb{R}[x]] \equiv (\lambda x.\mathbb{R}'[\mathtt{case}_A\ x\ e_1 \dots e_{|A|}])\ (\lambda y.y) \to_{\text{no}} \mathbb{R}'[\mathtt{case}_A\ (\lambda y.y)\ e_1 \dots e_{|A|}] \in IT$.

$\mathbb{R}[\cdot] \equiv \mathbb{R}'[[\cdot]\ r]$: We define $\mathbb{C}[\cdot] \overset{\text{def}}{=} (\lambda x.[\cdot])\ \mathtt{c}$ for a 0-ary constructor $\mathtt{c}$ and obtain $\mathbb{C}[\mathbb{R}[x]] \equiv (\lambda x.\mathbb{R}'[x\ r])\ \mathtt{c} \to_{\text{no}} \mathbb{R}[\mathtt{c}\ r] \in IT$.

□

Definition 2.72 (WT, well-typed). An expression, $s \in \Lambda$, is *well-typed*, iff s does not crash, that is iff

$$\forall t, r \in \Lambda : s \xrightarrow{*}_{\text{no}} t \wedge t \equiv \mathbb{R}[r] \implies r \notin DIT$$

For the set of well-typed expressions we also write WT.

The set of expressions, Λ, is thus partitioned into the sets WT and IT.

2.4.2 Properties of WT

Lemma 2.73. $s \in IT \implies s\Uparrow$.

Proof. Assume this does not hold, i.e. $\exists s : s \in IT \wedge s\not\Uparrow$. From the definition of IT we obtain $s \in IT \iff \exists t, r : s \xrightarrow{*}_{\text{no}} t \wedge t \equiv \mathbb{R}[r] \wedge r \in DIT$. s cannot normal-order reduce to $\mathbb{R}'[x]$ for a variable x, since neither is x ill-typed nor could there be any further normal-order reduction to an expression $\mathbb{R}[r]$ with $r \in DIT$ from $\mathbb{R}'[x]$ nor would there be a reduction context above $\mathbb{R}'[\cdot]$ with a directly ill-typed expression r' in the hole since $r' \not\equiv \mathbb{R}''[x]$ according to definition 2.66. So $s\Downarrow t$ would have to hold. The standardization

theorem says that for every WHNF t with $s \xrightarrow{*} t$ there is a WHNF t' with $s \xrightarrow{*}_{\text{no}} t'$. We assume t and t' to be WHNFs with $s \xrightarrow{*} t$ and $s \xrightarrow{*}_{\text{no}} t'$. Due to the uniqueness of the normal-order redex and since $s \in IT$ we conclude $s \xrightarrow{*}_{\text{no}} t'' \equiv \mathbb{R}[r] \xrightarrow{*}_{\text{no}} t'$ where $r \in DIT$. A directly ill-typed expression can never be a normal-order redex, thus there cannot be such a reduction of $\mathbb{R}[r]$ and neither can $\mathbb{R}[r]$ be a WHNF. □

Corollary 2.74. $s\Downarrow \implies s \in WT$.

Theorem 2.75 (Invariance of well-typedness). *Let $s, t \in \Lambda$ where $s \to t$, then $s \in WT \iff t \in WT$.*

Proof. Every one of the cases for DIT from definition 2.66 is covered in lemmata 2.64 and 2.65 so that the proof of theorem 2.59 can be transferred. □

There are expressions which are well-typed, but with which no well-typed applications can be formed, although they do have an FWHNF.

Lemma 2.76. *$\exists s \in WT : s$ has an FWHNF $\wedge \forall t \in \Lambda : s\ t \notin WT$.*

Proof. We provide an example. $s \overset{\text{def}}{=} \lambda x.\mathtt{c}\ s_1 \dots s_{\alpha(\mathtt{c})+1}$ for some $s_i \in \Lambda^0$. s is in FWHNF and thus in particular $s \in WT$. $s\ t \to_{\text{no}} \mathtt{c}\ s_1 \dots s_{\alpha(\mathtt{c})+1} \in DIT$. So for any $t : s\ t \notin WT$. □

In the Damas-Hindley-Milner type system there is usually no $\mathtt{case}_A$-constant defined for every type constructor A. For type checking purposes in that type system we may conceive of every $\mathtt{case}_A$ as a function (or a constructor) of type $(A\ T_1 \dots T_n) \to (T_{11} \to \dots \to T_{1m_1} \to B) \to \dots \to (T_{r1} \to \dots \to T_{rm_r} \to B)$, if $|A| = r$ and $\mathtt{c}_{A,i}$ is of type $T_{i1} \to \dots \to T_{im_i} \to (A\ T_1 \dots T_n)$.

Theorem 2.77. *$WT^{DHM} \subseteq WT$*

Proof. Expressions in WT^{DHM} do not give type errors during normal-order reduction and that is all we require for membership in WT. This is an immediate consequence of [Mil78, Theorem 1]. □

Well-typed expressions have the property that we can replace an arbitrary subexpression by a fresh variable or by `bot` and the resulting expression remains well-typed.

Lemma 2.78. *There are $v \in V$, $r \in \Lambda$, $\mathbb{C}[\cdot] \in \Lambda$ with $\mathbb{C}[r] \in WT \wedge \mathbb{C}[v] \in IT$.*

Proof. We define $\mathbb{C}[\cdot] \stackrel{\text{def}}{=} \texttt{case}_A\ ((\lambda v.[\cdot])\ (\lambda x.x))\ \underbrace{\texttt{1}\ldots\texttt{1}}_{|A|}$.
Obviously, for a 0-ary constructor $\texttt{c}_{A,i}$: $\mathbb{C}[\texttt{c}_{A,i}] \rightarrow_{\text{no}} \texttt{case}_A\ \texttt{c}_{A,i}\ \underbrace{\texttt{1}\ldots\texttt{1}}_{|A|} \rightarrow_{\text{no}} \texttt{1}$, but $\mathbb{C}[v] \rightarrow_{\text{no}} \texttt{case}_A\ (\lambda x.x)\ \underbrace{\texttt{1}\ldots\texttt{1}}_{|A|} \in DIT$. □

Lemma 2.79. *Let $r, \mathbb{C}[\cdot] \in \Lambda$, let $v \in V$ such that v is not bound in the hole of $\mathbb{C}[\cdot]$, then*

$$\mathbb{C}[r] \in WT \implies \mathbb{C}[v] \in WT.$$

Proof. Let $\mathbb{C}[v] \equiv \mathbb{R}^1[r^1] \rightarrow_{\text{no}} \ldots \rightarrow_{\text{no}} \mathbb{R}^n[r^n]$ where $r^n \in DIT$ be the normal-order reduction of $\mathbb{C}[v]$. Since v in not bound in $\mathbb{C}[\cdot]$ none of the normal-order reductions will bind anything to v. v can only be duplicated or can disappear. If present at all, v will be below argument depth 1 in any of the $\mathbb{R}^i[r^i]$. $\mathbb{C}[r]$ and $\mathbb{C}[v]$ do not differ above argument depth 1 and $\mathbb{C}[r] \equiv \mathbb{R}'^1[r'^1] \rightarrow_{\text{no}} \mathbb{R}'^2[r'^2]$ reduces the same position as $\mathbb{R}^1[r^1] \rightarrow_{\text{no}} \mathbb{R}^2[r^2]$. $\mathbb{R}^2[r^2]$ and $\mathbb{R}'^2[r'^2]$

do not differ above argument depth 1 since in all the $\mathbb{R}^i[r^i]$ v does not occur above argument depth 1. The argument can be extended to $\mathbb{R}^n[r^n]$ and $\mathbb{R}'^n[r'^n]$ and since v does not occur above argument depth 1 in $\mathbb{R}^n[r^n]$, r'^n must also belong to DIT. The latter can be seen since $r^n \equiv \texttt{case}_A\ (\texttt{c}_{A'}\ t_1 \ldots t_{\alpha(\texttt{c}_{A'})})\ e_1 \ldots e_{|A|}$ with $A' \neq A$ implies $r'^n \equiv \texttt{case}_A\ (\texttt{c}_{A'}\ t'_1 \ldots t'_{\alpha(\texttt{c}_{A'})})\ e'_1 \ldots e'_{|A|}$, $r^n \equiv \texttt{case}_A\ e\ e_1 \ldots e_{|A|}$ with an FWHNF e implies $r'^n \equiv \texttt{case}_A\ e'\ e'_1 \ldots e'_{|A|}$ with an FWHNF e' and $r^n \equiv \texttt{c}\ t_1 \ldots t_n$ with $n > \alpha(\texttt{c})$ implies $r'^n \equiv \texttt{c}\ t'_1 \ldots t'_n$, if r^n and r'^n do not differ above argument depth 1. □

Lemma 2.80. *Let $\mathbb{C}[\cdot] \in \Lambda$ and $v \in V$ where v is not bound in $\mathbb{C}[\cdot]$, then*

$$\mathbb{C}[v] \in WT \implies \mathbb{C}[\texttt{bot}] \in WT.$$

Proof. In the normal-order reduction of $\mathbb{C}[\texttt{bot}] \rightarrow_{\text{no}} \ldots \rightarrow_{\text{no}} \mathbb{R}^n[r^n]$ with $r^n \in DIT$ `bot` will never appear in a reduction context. It suffices to show:

> If none of the holes of a context $\mathbb{D}[\ldots]$ is in a reduction context and $\mathbb{D}[\underbrace{\texttt{bot},\ldots,\texttt{bot}}_{n}] \rightarrow_{\text{no}} \mathbb{E}[\underbrace{\texttt{bot},\ldots,\texttt{bot}}_{m}]$, then $\mathbb{D}[\underbrace{v,\ldots,v}_{n}] \rightarrow_{\text{no}} \mathbb{E}[\underbrace{v,\ldots,v}_{m}]$.

If none of the holes of $\mathbb{D}[\ldots]$ is inside the normal-order redex of $\mathbb{D}[\texttt{bot},\ldots,\texttt{bot}] \equiv \mathbb{R}[r]$, then $\mathbb{D}[v,\ldots,v] \equiv \mathbb{R}'[r]$ and obviously the same reduction can be applied. That $\mathbb{R}'[\cdot]$ is indeed a reduction context can easily be proved by induction.
If one of the holes of $\mathbb{D}[\ldots]$ is inside the normal-order redex of $\mathbb{D}[\texttt{bot},\ldots,\texttt{bot}] \equiv \mathbb{R}[r]$, then either $r \equiv \texttt{case}_A\ \texttt{c}_{A,i}\ t_1 \ldots t_{\alpha(\texttt{c}_{A,i})}\ e_1 \ldots e_{|A|}$ or $r \equiv (\lambda x.s)\ t$. In the first case

the holes can be in the t_i or the e_i and it is obvious that $\mathbb{D}[v,\dots,v]$ can be reduced just as $\mathbb{D}[\texttt{bot},\dots,\texttt{bot}]$ to obtain $\mathbb{E}[v,\dots,v]$. In the second case we can also reduce $\mathbb{D}[v,\dots,v]$ just as $\mathbb{D}[\texttt{bot},\dots,\texttt{bot}]$ and since v is not bound in any of the holes of $\mathbb{D}[\dots]$ we get $\mathbb{D}[v,\dots,v] \to_{\text{no}} \mathbb{E}[v,\dots,v]$. □

Corollary 2.81. $\mathbb{C}[r] \in WT \implies \mathbb{C}[\texttt{bot}] \in WT$.

Proof. Lemmata 2.80 and 2.79. □

With the lemma below we obtain the possibility to replace subexpressions in well-typed expressions by other well-typed expressions reducing to the same WHNF to obtain well-typed expressions.

Lemma 2.82. *Let $s \to t$ then*

$$\mathbb{D}[s] \in WT \iff \mathbb{D}[t] \in WT.$$

Proof. If $s \to t$ then $\mathbb{D}[s] \to \mathbb{D}[t]$ and the statement follows from the invariance of well-typedness. □

2.4.3 Undecidability

We will now show that being well-typed is an undecidable property. The proof is by reduction of the halting problem for Turing-machines. We will not present the reduction in its entirety but only reduce the halting problem for Λ to the type check problem.

Definition 2.83. The *halting problem for* Λ consists of deciding for a closed, well-typed Λ-expression t, if $t\not\Downarrow$ or $t\Downarrow$.

Definition 2.84. The *type-check problem* consists of deciding for a Λ-expression t, if $t \in WT$ or $t \in IT$.

We reduce the halting problem for Λ to the type check problem by creating the expression $\mathbb{R}[h]$ for any instance h of the halting problem. If $\mathcal{A} = \{A_1,\dots,A_m\}$ are the type-constructors of which constructors appear in h, we define $\mathbb{R}[\cdot] \overset{\text{def}}{=} \texttt{case}_A\ [\cdot]\ \underbrace{\texttt{bot}\dots\texttt{bot}}_{|A|}$ for some $A \notin \mathcal{A}$. $\mathbb{R}[\cdot]$ is a reduction context and $\mathbb{R}[h]$ is an instance of the type-check problem.

Lemma 2.85. $\mathbb{R}[h] \in IT \iff h\Downarrow$.

Proof. We show the two directions separately.

$\Longleftarrow$: Assume $h\Downarrow$. If h converges to an SCWHNF t, t will have a top level constructor not in A and $\mathbb{R}[t] \in DIT$. If h converges to an FWHNF t, $\mathbb{R}[t]$ is in DIT also.

$\neg \Longleftarrow \neg$: Assume $h\not\Downarrow$. Either $h\Uparrow$ and we can continue to reduce in normal order without ever reaching a WHNF and thus the condition for $\mathbb{R}[h] \in IT$ is not satisfied. Further normal-order reduction will always be possible since otherwise $h \in IT$. Or $h \xrightarrow{*}_{\text{no}} \mathbb{R}[x]$ and no further reduction is possible, but $x \notin DIT$. □

Lemma 2.86. *Diverging but well-typed expressions have infinite normal-order reductions.*

Proof. Assume this does not hold. Then there is a diverging but well-typed expression t, which cannot be further normal-order reduced after finitely many normal-order reductions. Since t is well-typed, we conclude that after finitely many normal-order reductions we have reached a WHNF and thus t is not diverging, in contradiction to our assumption. □

2.5 Contextual semantics

The meaning of expressions in the *contextual* semantics is their equivalence class of the contextual equivalence. The latter regards two expressions as equivalent if their insertion as a subexpression into an arbitrary program yields identical termination behavior for both (cf. 2.87). In this way the equivalence classes of the contextual equivalence correspond to the elements of the semantic domains in denotational semantics.
An important advantage of the contextual semantics is that it is always fully abstract for the observation of termination if it is adequate. For denotational semantics this property is difficult to obtain [Ong95, AMJ94].
We discuss adequacy and full abstraction in section 2.5.5.

2.5.1 Contextual order

Definition 2.87 ($\leq_c$, contextual order, contextual approximation). Let s, t be (possibly open) expressions.

$$s \leq_c t \iff \forall \mathbb{C}[\cdot] : \mathbb{C}[s]\Downarrow \implies \mathbb{C}[t]\Downarrow \tag{2.4}$$

$$s \equiv_c t \iff s \leq_c t \wedge t \leq_c s \tag{2.5}$$

$\leq_c$ is called *contextual order* or *contextual approximation* and $\equiv_c$ is called *contextual equivalence.*

The contextual order is a definition central to this work. We will also frequently speak about expressions which are not so related. From definition 2.87 it follows immediately that this is the case exactly when there is a context which converges when applied to one argument and diverges for the other. Thus we arrive at the definition:

Definition 2.88. Let $s, t \in \Lambda$ and let $\mathbb{C}[\cdot]$ be a context. If $\mathbb{C}[s]\Downarrow$ and $\mathbb{C}[t]\Uparrow$ we say that $\mathbb{C}[\cdot]$ *exposes the difference between* s and t, $\mathbb{C}[\cdot]$ *differentiates between* s and t or $\mathbb{C}[\cdot]$ *distinguishes* s *from* t.

Lemma 2.89. *$\leq_c$ is a pre-order on Λ.*

Proof. We need to show that $\leq_c$ is reflexive and transitive.

reflexive: obvious.

transitive: Let $s \leq_c t \leq_c u$, we show $s \leq_c u$. Let $\mathbb{C}[\cdot]$ be an arbitrary context. If $\mathbb{C}[s]\not\Downarrow$ the condition is satisfied. If $\mathbb{C}[s]\Downarrow$, then $\mathbb{C}[t]\Downarrow$ and then $\mathbb{C}[u]\Downarrow$. Thus $\mathbb{C}[s]\Downarrow \implies \mathbb{C}[u]\Downarrow$ holds also. □

Lemma 2.90. *Every context is monotonous with respect to $\leq_c$ on Λ-expressions.*

Proof. Let $s, t \in \Lambda$. $s \leq_c t \iff \forall \mathbb{C}[\cdot] : \mathbb{C}[s]\Downarrow \implies \mathbb{C}[t]\Downarrow$. Since this must hold for all $\mathbb{C}[\cdot]$, it must also hold for arbitrarily chosen $\mathbb{D}[\mathbb{E}[\cdot]]$. Thus $\forall \mathbb{D}[\cdot], \mathbb{E}[\cdot] : \mathbb{D}[\mathbb{E}[s]]\Downarrow \implies \mathbb{D}[\mathbb{E}[t]]\Downarrow$ and so $\mathbb{E}[s] \leq_c \mathbb{E}[t]$. □

Corollary 2.91. *$\leq_c$ is a precongruence and $\equiv_c$ is a congruence.*

Corollary 2.92. *Let $s_1, \ldots, s_n, t_1, \ldots, t_n \in \Lambda$ with $s_1 \leq_c t_1, \ldots, s_n \leq_c t_n$ and let $\mathbb{C}[\cdots]$ be a multi-hole context, then*

$$\mathbb{C}[s_1, \ldots, s_n] \leq_c \mathbb{C}[t_1, \ldots, t_n].$$

Proof. We can define contexts $\mathbb{D}^1[\cdot] \equiv \mathbb{C}[\cdot, s_2, \ldots, s_n], \mathbb{D}^2[\cdot] \equiv \mathbb{C}[t_1, \cdot, s_3, \ldots, s_n], \ldots, \mathbb{D}^n[\cdot] \equiv \mathbb{C}[t_1, \ldots, t_{n-1}, \cdot]$. The statement follows from lemma 2.90, because every one of the $\mathbb{D}^i[\cdot]$ has one hole and for every $1 \leq i < n : \mathbb{D}^i[t_i] \equiv \mathbb{D}^{i+1}[s_{i+1}]$. Thus $\mathbb{C}[s_1, \ldots, s_n] \equiv \mathbb{D}^1[s_1] \leq_c \mathbb{D}^1[t_1] \equiv \mathbb{D}^2[s_2] \leq_c \ldots \leq_c \mathbb{D}^n[t_n] \equiv \mathbb{C}[t_1, \ldots, t_n]$. □

Contexts can be viewed as *challenges*, and divergence and convergence are the *observables*. Consequently, the contextual order can be seen as a *measurement of information content*: the more challenges for which we can observe a value, the more information the expression contains.
Theorem 2.93 states that reduction does not change information content.
Intuitively, *trivial challenges* would be the contexts, that do not vary with their hole. There are *non-trivial challenges*, i.e. contexts, which do differentiate some expressions, that do not differentiate some others although they could be differentiated (cf. lemma 2.94).

Theorem 2.93. *Let $s, t \in \Lambda$, $s \to t$, then $s \equiv_c t$.*

Proof. $\forall \mathbb{C}[\cdot] : \mathbb{C}[s] \to \mathbb{C}[t]$ and with theorem 2.59 we conclude that $\mathbb{C}[s]\Downarrow \iff \mathbb{C}[t]\Downarrow$. □

The converse of lemma 2.90 does not hold, i.e. we cannot conclude from the $\leq_c$ relation between expressions that their subexpressions are so related.

Lemma 2.94. *$\mathbb{C}[u] \leq_c \mathbb{C}[v]$ does not imply $u \leq_c v$, not even if $\exists s, t : \mathbb{C}[s] \not\equiv_c \mathbb{C}[t]$.*

Proof. As a counter-example we use a type constructor A with two unary constructors $\texttt{c}_{A,1}$ and $\texttt{c}_{A,2}$ to define. $\mathbb{C}[\cdot] \overset{\text{def}}{=} \texttt{case}_A\ [\cdot]\ \texttt{id (K [])}$ Then $\mathbb{C}[\texttt{c bot}] \not\equiv_c \mathbb{C}[\texttt{c []}]$ but $\mathbb{C}[\texttt{d []}] \leq_c \mathbb{C}[\texttt{d bot}]$ although $\texttt{d []} \not\leq_c \texttt{d bot}$. □

Intuitively, this is possible since $\mathbb{C}[\cdot]$ may maintain the order among some elements while it smashes the order between others. We may however prove this property for constructors and λ bindings.

Lemma 2.95. $\texttt{c}_{A,i}\ s_1 \dots s_{\alpha(\texttt{c}_{A,i})} \leq_c \texttt{c}_{A,i}\ t_1 \dots t_{\alpha(\texttt{c}_{A,i})} \iff \forall i : s_i \leq_c t_i$.

Proof.

$\Longrightarrow$: Assume this would not hold. There must be some j for which $s_j \not\leq_c t_j$. For $\mathbb{C}[\cdot] \overset{\text{def}}{=} \texttt{sel}_{A,i,j}\ [\cdot]$ we obtain $\mathbb{C}[\texttt{c}_{A,i}\ s_1 \dots s_{\alpha(\texttt{c}_{A,i})}] \not\leq_c \mathbb{C}[\texttt{c}_{A,i}\ t_1 \dots t_{\alpha(\texttt{c}_{A,i})}]$. Let $\mathbb{D}[\cdot]$ be the context witnessing $s_j \not\leq_c t_j$. $\mathbb{D}[\mathbb{C}[\cdot]]$ witnesses $\texttt{c}_{A,i}\ s_1 \dots s_{\alpha(\texttt{c}_{A,i})} \not\leq_c \texttt{c}_{A,i}\ t_1 \dots t_{\alpha(\texttt{c}_{A,i})}$ and we have a contradiction. The assumption is false and the claim proved.

$\Longleftarrow$: since $\leq_c$ is a precongruence. □

Proposition 2.96. *Let $s, t \in \Lambda$, then*

$$s \leq_c t \iff \lambda x.s \leq_c \lambda x.t.$$

Proof.

$\Longrightarrow$: Since $\leq_c$ is a precongruence.

$\Longleftarrow$: Assume $\lambda x.s \leq_c \lambda x.t$ and let $\mathbb{C}[\cdot]$ be an arbitrary context with $\mathbb{C}[s]\Downarrow$. We define the context $\mathbb{D}[\cdot] \overset{\text{def}}{=} \mathbb{C}[(\lambda x.[\cdot])\ x]$. Obviously, $\mathbb{D}[s] \to \mathbb{C}[s]$. The definition of convergence implies $\mathbb{D}[s]\Downarrow$. We apply the precongruence of $\leq_c$ and the premise to substantiate $\mathbb{D}[t] \equiv \mathbb{C}[(\lambda x.t)\ x]\Downarrow$. By theorem 2.59 $\mathbb{C}[(\lambda x.t)\ x] \to \mathbb{C}[t]$ implies $\mathbb{C}[t]\Downarrow$. □

Remark 2.97. $s \leq_c t \implies ((s \in WT \wedge s\Downarrow) \implies (t \in WT \wedge t\Downarrow))$.

2.5.2 Context Lemma

In the definition 2.87 the quantification is over all contexts. This would be unwieldy for many proofs and one would like to restrict attention to only some specific contexts. This idea is not new: commonly a *context lemma* is stated in works on operational semantics of λ-calculi [Mil77, San98a, MS99]. This lemma then states that it suffices to focus ones attention to reduction contexts to prove two expressions to be contextually ordered. We provide a range of results that allow to focus on different kinds of contexts. The following lemma is a first step in this direction as it allows to consider closing contexts only.

Lemma 2.98. *Let $s, t \in \Lambda$, then*

$$s \leq_c t \iff \forall \mathbb{C}[\cdot] : \mathbb{C}[s] \textit{ and } \mathbb{C}[t] \textit{ are closed } \wedge \mathbb{C}[s]\Downarrow \implies \mathbb{C}[t]\Downarrow.$$

Proof.

$\implies$: obvious.

$\neg \implies \neg$: We know $s \not\leq_c t \iff \exists \mathbb{C}[\cdot] : \mathbb{C}[s]\Downarrow \wedge \mathbb{C}[t]\not\Downarrow$.

If $\mathbb{C}[s]$ and $\mathbb{C}[t]$ are closed the statement holds.

Let $\mathcal{FV}(\mathbb{C}[s]) \cup \mathcal{FV}(\mathbb{C}[t]) = \{x_1, \ldots, x_n\}$, then we define $\mathbb{D}[\cdot] \overset{\text{def}}{=} (\lambda x_1 \ldots x_n.\mathbb{C}[\cdot])\ \underbrace{\texttt{bot}\ldots\texttt{bot}}_{n}$. Now $\mathbb{D}[s]$ and $\mathbb{D}[t]$ are closed. Both $\mathbb{D}[s]\Downarrow$ and $\mathbb{D}[t]\Uparrow$ are proved separately.

$\mathbb{D}[s]\Downarrow$: We can reduce $\mathbb{D}[s] \xrightarrow{*}_{\text{no}} \mathbb{C}[s][\overrightarrow{\texttt{bot}/x}]$. Since reduction sequences are stable with respect to substitution of free variables, we can apply the reduction sequence from $\mathbb{C}[s]$ to WHNF to $\mathbb{C}[s][\overrightarrow{\texttt{bot}/x}]$.

$\mathbb{D}[t]\Uparrow$: Here we also argue with the stability of reduction sequences with respect to substitution. If $\mathbb{C}[t]$ normal-order reduces infinitely or if $\mathbb{C}[t]$ normal-order reduces to an expression in *DIT* then so does $\mathbb{D}[t]$. If $\mathbb{C}[t]$ reduces to $\mathbb{R}[x]$ for some reduction context $\mathbb{R}[\cdot]$ and some variable x, then x is free in $\mathbb{C}[t]$ due to lemma 2.30 thus $\mathbb{D}[t] \xrightarrow{*}_{\text{no}} \mathbb{C}[t][\overrightarrow{\texttt{bot}/x}] \xrightarrow{*}_{\text{no}} \mathbb{R}'[\texttt{bot}]$, which diverges. □

Conversely, if the reduction relation would not be stable with respect to substitution it would be easy to see that the equivalence of lemma 2.98 would not hold. As an admittedly contrived example consider a hypothetical language in which $(x\ \texttt{bot}) \to \texttt{bot}$ even if x is free, but substituting a lazy constructor would produce a WHNF.
If the two expressions are closed it suffices to consider all closed reduction contexts. This is a result that is commonly found in the literature using operational techniques in semantics, e.g. [Kut00, San98a].

Lemma 2.99 (Context Lemma). *Let $s, t \in \Lambda^0$, i.e. s and t are closed Λ-expressions, then*

$$s \leq_c t \iff \forall \mathbb{R}[\cdot] : \mathbb{R}[s], \mathbb{R}[t] \in \Lambda^0 \wedge \mathbb{R}[s]\Downarrow \implies \mathbb{R}[t]\Downarrow.$$

Proof. We prove the more general statement:

Let $s_i, t_i \in \Lambda^0$ for which for all reduction contexts $\mathbb{R}[\cdot] : \mathbb{R}[s_i], \mathbb{R}[t_i] \in \Lambda^0 \wedge \mathbb{R}[s_i]\Downarrow \implies \mathbb{R}[t_i]\Downarrow$ then $\forall \mathbb{C}[\ldots]$ for which $\mathbb{C}[s_1, \ldots, s_n]$ and $\mathbb{C}[t_1, \ldots, t_n]$ are closed $\mathbb{C}[s_1, \ldots, s_n]\Downarrow \implies \mathbb{C}[t_1, \ldots, t_n]\Downarrow$.

Assume this does not hold, then for every reduction context $\mathbb{R}[\cdot]$ and every $i : \mathbb{R}[s_i]$ and $\mathbb{R}[t_i]$ are closed and $\mathbb{R}[s_i]\Downarrow$ implies $\mathbb{R}[t_i]\Downarrow$, but there is a multi-context $\mathbb{C}[\ldots] : \mathbb{C}[\vec{s}]\Downarrow \wedge \mathbb{C}[\vec{t}]\not\Downarrow$. We choose the smallest context among these with respect to the lexicographic order on pairs having as components

1. the number of normal-order steps from $\mathbb{C}[\vec{s}]$ to WHNF and

2. the number of holes in $\mathbb{C}[\dots]$.

There are two cases:

1. One of the holes of $\mathbb{C}[\dots]$ is in a reduction context. I.e. $\exists j : \mathbb{C}[s_1, \dots, s_{j-1}, \cdot, s_{j+1}, \dots, s_n]$ (and thus $\mathbb{C}[t_1, \dots, t_{j-1}, \cdot, t_{j+1}, \dots, t_n]$) is a reduction context. We define $\mathbb{D}[\cdot_1, \dots, \cdot_{n-1}] \overset{\text{def}}{=} \mathbb{C}[\cdot_1, \dots, \cdot_{j-1}, s_j, \cdot_j, \dots, \cdot_{n-1}]$. Since the number of holes in $\mathbb{D}[\dots]$ is less than the number of holes in $\mathbb{C}[\dots]$ and the number of normal-order reductions for $\mathbb{D}[s_1, \dots, s_{j-1}, s_{j+1}, \dots, s_n] \equiv \mathbb{C}[\vec{s}]$ remains unchanged, and since $\mathbb{C}[\dots]$ was chosen to be minimal, we obtain $\mathbb{D}[t_1, \dots, t_{j-1}, t_{j+1}, \dots, t_n] \equiv \mathbb{C}[t_1, \dots, t_{j-1}, s_j, t_{j+1}, \dots, t_n]\Downarrow$.
   $\mathbb{C}[t_1, \dots, t_{j-1}, \cdot, t_{j+1}, \dots, t_n]$ is a reduction context and thus the premise implies $\mathbb{C}[\vec{t}]\Downarrow$ in contradiction to the assumption.

2. None of the holes of $\mathbb{C}[\dots]$ is in a reduction context, then either $\mathbb{C}[\vec{s}]$ is a WHNF and it follows from proposition 2.35 that $\mathbb{C}[\vec{t}]$ is a WHNF as well or there is a normal-order reduction of $\mathbb{C}[\vec{s}]$ with a normal-order redex having its root above the holes. With the same proposition we substantiate that the identical normal-order reduction can be performed for $\mathbb{C}[\vec{t}]$.

   Since neither the s_i nor the t_i have variables occurring free the normal-order reductions cannot substitute anything into the expressions. We can thus locate the unchanged s_i and t_i in the respective contractum, possibly duplicated. Hence $\mathbb{C}[\vec{s}] \to_{\text{no}} \mathbb{E}[s_{i_1}, \dots, s_{i_m}]$ where $i_j \in \{1, \dots, n\}$ and $\mathbb{C}[\vec{t}] \to_{\text{no}} \mathbb{E}[t_{i_1}, \dots, t_{i_m}]$. Since $\mathbb{C}[\dots]$ was chosen to be minimal $\mathbb{C}[\vec{t}]\Downarrow$ follows and we have a contradiction to the assumption. □

We have formulated and proved the context lemma for closed expressions only. With the following lemmata we provide some means to employ it for open expressions also.

Lemma 2.100. *Let $s, t \in \Lambda$, then*

$$s \leq_c t \iff \forall \sigma : \sigma s, \sigma t \in \Lambda^0 \implies \sigma s \leq_c \sigma t.$$

Proof.

$\Longrightarrow$: This follows from the precongruence of $\leq_c$ and theorem 2.59.

$\Longleftarrow$: Without loss of generality there are variables occurring free in s or t, otherwise no proof is needed. Suppose $\{x_1, \dots, x_n\} = \mathcal{FV}(s) \cup \mathcal{FV}(t)$.

$$\begin{aligned} & \forall \sigma : \sigma s, \sigma t \in \Lambda^0 \implies \sigma s \leq_c \sigma t \\ \Longrightarrow\ & (\sigma \text{ is a ground substitution } + \text{ theorem 2.59}) \\ & \forall \sigma : \sigma s, \sigma t \in \Lambda^0 \implies (\lambda \vec{x}.s)\ \sigma\vec{x} \leq_c (\lambda \vec{x}.t)\ \sigma\vec{x} \\ \Longrightarrow\ & (\text{pre-extensionality of } \leq_c) \\ & \forall \sigma : \sigma s, \sigma t \in \Lambda^0 \implies \lambda x_1 \dots x_n.s \leq_c \lambda x_1 \dots x_n.t \\ \Longrightarrow\ & (\text{proposition 2.96}) \\ & s \leq_c t. \quad \square \end{aligned}$$

In conjunction with the context lemma for closed expressions we can then state the following corollary, which corresponds to the CIU theorem in [MST96].

Corollary 2.101. *Let $s, t \in \Lambda$, then*

$$s \leq_c t \iff \forall \mathbb{R}[\cdot] \in \Lambda^0, \sigma : \sigma s, \sigma t \in \Lambda^0 \implies (\mathbb{R}[\sigma s]\Downarrow \implies \mathbb{R}[\sigma t]\Downarrow). \tag{2.6}$$

We may restrict our attention to an even simpler kind of expressions when employing the context lemma for open expressions.

Lemma 2.102. *Let $s, t \in \Lambda$, let $x_1, \ldots, x_n$ be all the variables occurring free in s or t and let $s' \overset{def}{=} \lambda x_1 \ldots x_n.s$ and $t' \overset{def}{=} \lambda x_1 \ldots x_n.t$, then*

$$s \leq_c t \iff \forall \mathbb{R}[\cdot] \in \Lambda^0 : \mathbb{R}[s']\Downarrow \implies \mathbb{R}[t']\Downarrow.$$

Proof. From proposition 2.96 we get $s \leq_c t \iff s' \leq_c t'$. s' and t' are closed. Application of the context lemma proves the statement. □

Intuitively, lemma 2.102 also implies corollary 2.101, since the quantification over all the reduction contexts in the first also captures those which apply the hole and thus s' or t' to closed arguments.

2.5.3 Normalizing higher-order expressions

It suffices to apply two functions to arbitrary Λ-expressions to see that they are $\leq_c$-related.

Proposition 2.103 (pre-extensionality). *Let $f, g \in \Lambda$ with $f\Downarrow_F$ and $g\Downarrow_F$, then*

$$\forall a \in \Lambda : f\ a \leq_c g\ a \iff f \leq_c g.$$

Proof.

$\Longleftarrow$: Since $\leq_c$ is a precongruence.

$\Longrightarrow$: Assume this does not hold then $\forall a : f\ a \leq_c g\ a$, but $f \not\leq_c g$. Suppose $\{x_1, \ldots, x_n\} = \mathcal{FV}(f) \cup \mathcal{FV}g$, $f' \overset{\text{def}}{=} \lambda x_1 \ldots x_n.f$

and $g' \overset{\text{def}}{=} \lambda x_1 \ldots x_n.g$. With lemma 2.100 we conclude

$$\forall a, \sigma : \sigma(f\ a), \sigma(g\ a) \in \Lambda^0 \implies \sigma(f\ a) \leq_c \sigma(g\ a) \quad (2.7)$$

and with lemma 2.102 we conclude

$$\exists \mathbb{R}[\cdot] \in \Lambda^0 : \mathbb{R}[f']\Downarrow \wedge \mathbb{R}[g']\Uparrow. \quad (2.8)$$

$\mathbb{R}[\cdot]$ must apply the hole to some expression, because

- $\mathbb{R}[\cdot] \equiv [\cdot] \implies \mathbb{R}[g']\Downarrow$ and
- $\mathbb{R}[\cdot] \equiv \mathbb{S}[\texttt{case}_A\ [\cdot]\ \ldots] \implies \mathbb{R}[f']\Uparrow$,

which is both in contradiction to (2.8). Hence $\mathbb{R}[\cdot] \equiv \mathbb{S}^1[[\cdot]\ e_1]$ and thus

$$\mathbb{R}[f'] \rightarrow_{\text{no}} \mathbb{S}^1[\lambda x_2 \ldots x_n.f[^{e_1}/_{x_1}] \text{ and}$$
$$\mathbb{R}[g'] \rightarrow_{\text{no}} \mathbb{S}^1[\lambda x_2 \ldots x_n.g[^{e_1}/_{x_1}].$$

Since $\mathbb{R}[\cdot] \in \Lambda^0$ no renaming of variables is necessary for e_1 since no name capture can occur. By repeating the argument above we reach $\mathbb{S}^n[\cdot] \equiv \mathbb{S}^{n+1}[[\cdot]\ e]$ with

$$\mathbb{R}[f'] \xrightarrow{*}_{\text{no}} \mathbb{S}^{n+1}[f[^{e_1}/_{x_1}] \ldots [^{e_n}/_{x_n}]\ e] \text{ and}$$
$$\mathbb{R}[g'] \xrightarrow{*}_{\text{no}} \mathbb{S}^{n+1}[g[^{e_1}/_{x_1}] \ldots [^{e_n}/_{x_n}]\ e].$$

As before $e \in \Lambda^0$ and we can apply (2.7) and theorem 2.59 to obtain $\mathbb{R}[f']\Downarrow \implies \mathbb{R}[g']\Downarrow$ in contradiction to (2.8). The implication follows. □

The statement of proposition 2.103 is not true for arbitrary Λ-expressions f and g since application to an argument could produce an ill-typed expression. This is the case even for some f, g which are $\leq_c$-related.

Example 2.104. Consider $f{\Downarrow_S}\texttt{bot : []}$ and $g{\Downarrow_S}\texttt{bot : bot}$, then $\forall a \in \Lambda : f\ a \leq_c g\ a$ but $f \not\leq_c g$ and $g \leq_c f$.

In our definition 2.36 FWHNFs may be of different kinds: they can be abstractions and they may be unsaturated constructor- or $\texttt{case}_A$-applications. We show now that the FWHNFs are exactly the expressions contextually equivalent to abstractions.

Lemma 2.105. *Let $s \in \Lambda$ with $s{\Downarrow_F}$, then $s \equiv_c \lambda x.(s\ x)$.*

Proof. Obviously, $\lambda x.(s\ x)$ is an FWHNF. If we assume that $s \not\equiv_c \lambda x.(s\ x)$ then by proposition 2.103 $\exists a : s\ a \not\equiv_c (\lambda x.(s\ x))\ a$. According to theorem 2.93 $(\lambda x.(s\ x))\ a \equiv_c s\ a$ contradicting the assumption. This proves the statement. □

Corollary 2.106. *Let $s \in \Lambda$ with $s{\Downarrow_F}$, then $\exists t \in \Lambda : s \equiv_c \lambda x.t$.*

Expressions which do not have an FWHNF are not equivalent to any abstraction.

Lemma 2.107. *Let $s \in \Lambda$ with $s \not\Downarrow_F$, then $\forall t \in \Lambda : s \not\equiv_c \lambda x.t$.*

Proof. $s{\not\Downarrow_F} \implies s{\Uparrow} \vee s{\Downarrow_S}\texttt{c}_{A,i}\ e_1 \dots e_{\alpha(\texttt{c}_{A,i})} \vee s \xrightarrow{*} \mathbb{R}[x]$. In the first case $[\cdot]$ distinguishes s from $\lambda x.t$, in the second $\texttt{case}_A\ [\cdot]\ \underbrace{\texttt{1}\dots\texttt{1}}_{|A|}$ is such a context and in the third for any of the contexts from lemma 2.71 $\mathbb{C}[s] \in IT$ but $\mathbb{C}[\lambda x.t] \rightarrow_{\text{no}} \lambda x.t$. □

2.5.4 Structure

The contextual order imposes a structure upon Λ which will be presented in this section.
The divergent expressions are the least expressions in the contextual order. Furthermore all divergent expressions are equivalent

and from now on we can use `bot` as a representative of all diverging expressions.

Proposition 2.108. *Let $s \in \Lambda$, then*

$$\forall s : s{\Uparrow} \iff \forall t : s \leq_c t.$$

Proof.

$\neg \implies \neg$: If $s{\Downarrow}$ then $s \not\leq_c \texttt{bot}$.

$\implies$: By corollary 2.101 $s \leq_c t \iff \forall \mathbb{R}[\cdot] \in \Lambda^0, \sigma : \sigma s, \sigma t \in \Lambda^0 \implies (\mathbb{R}[\sigma s]{\Downarrow} \implies \mathbb{R}[\sigma t]{\Downarrow})$. Let $\mathbb{R}[\cdot]$ be an arbitrary closed reduction context and σ a closing substitution for s and t. Proposition 2.35 implies $\sigma s{\Uparrow}$. Then there cannot be a reduction to WHNF of $\mathbb{R}[\sigma s]$ and the implication is satisfied for every t. □

Corollary 2.109. *Let $s, t \in \Lambda$ with $s{\Uparrow}$ and $t{\Uparrow}$ then $s \equiv_c t$.*

Lemma 2.110. *Let t, t_i, t'_i be Λ-expressions, $\texttt{c}, \texttt{c}'$ be constructors and $n \leq \alpha(\texttt{c}), m \leq \alpha(\texttt{c}')$. Then $t{\Downarrow}\texttt{c}\ t_1 \dots t_n \wedge t{\Downarrow}\texttt{c}'\ t'_1 \dots t'_m \implies \texttt{c} = \texttt{c}'$ and $m = n$.*

Proof. Due to transitivity of $\equiv_c$ and theorem 2.93 $\texttt{c}\ t_1 \dots t_n \equiv_c \texttt{c}'\ t'_1 \dots t'_m$. Assume $\texttt{c} \neq \texttt{c}'$ or $m \neq n$.

1. In the first case $\exists A, i : \texttt{c} \equiv \texttt{c}_{A,i}$ and we use a context

$$\mathbb{C}[\cdot] \overset{\text{def}}{=} \texttt{case}_A\ ([\cdot]\ \underbrace{\texttt{bot}\dots\texttt{bot}}_{\alpha(\texttt{c})-n})\ \underbrace{\texttt{bot}\dots\texttt{bot}}_{i-1}\ \texttt{1}\ \underbrace{\texttt{bot}\dots\texttt{bot}}_{|A|-i}$$

to exhibit the contradiction $\mathbb{C}[t] \xrightarrow{*} \mathbb{C}[\texttt{c}\ t_1 \dots t_n]{\Downarrow}$ and $\mathbb{C}[t] \xrightarrow{*} \mathbb{C}[\texttt{c}'\ t'_1 \dots t'_m]{\Uparrow}$.

2. In the second case we may assume $\mathtt{c} \equiv \mathtt{c}'$. Here also do we easily recognize that $\mathbb{C}[\mathtt{c}\ t_1 \dots t_n]\Downarrow$ but $\mathbb{C}[\mathtt{c}'\ t'_1 \dots t'_m]\Uparrow$ for either case: $m < n$ and $m > n$. □

If an expression obtained from setting a non-converging expression into a context converges, then any expression obtained with this context will converge.

Lemma 2.111. *$s\not\Downarrow \wedge \mathbb{C}[s]\Downarrow \implies \mathbb{C}[t]\Downarrow$ for any $t \in \Lambda$.*

Proof. $\forall t \in \Lambda : s \leq_c t$ and since $\leq_c$ is a precongruence $\forall \mathbb{C}[\cdot] : \mathbb{C}[s]\Downarrow \implies \mathbb{C}[t]\Downarrow$. □

We clarify the relation of some kinds of Λ-expressions, such as those having a WHNF etc. to obtain the diagram of figure 2.3.

Lemma 2.112. *Let $s, t, t_i \in \Lambda$ and let x, y be variables, then*

1. $\lambda x.\mathtt{bot} \not\leq_c \mathtt{bot}$
2. $\lambda x.\mathtt{bot} \leq_c \mathtt{c}\ t_1 \dots t_{\alpha(\mathtt{c})}$
3. $\forall n \geq 2 : \lambda x_1 \dots x_n.\mathtt{bot} \not\leq_c \mathtt{c}\ t_1 \dots t_{\alpha(\mathtt{c})}$
4. $\forall n : \lambda x_1 \dots x_n.\mathtt{bot} \leq_c \lambda x_1 \dots x_{n+1}.\mathtt{bot}$
5. *s has an SCWHNF implies $s \not\leq_c \lambda x.t$*
6. *s has a WHNF and $t \xrightarrow{*}_{no} \mathbb{R}[x] \implies s \not\leq_c t$*
7. $x \not\equiv y \wedge s \xrightarrow{*}_{no} \mathbb{R}[x] \wedge t \xrightarrow{*}_{no} \mathbb{R}'[y] \implies s \not\leq_c t$

Proof. It will suffice to consider closed reduction contexts and closing substitutions.

1. Obviously, $\lambda x.\mathtt{bot}\Downarrow \wedge \mathtt{bot}\Uparrow$.

2. Assume $\exists \mathbb{R}[\cdot], \sigma : \mathbb{R}[\lambda x.\mathtt{bot}]\Downarrow \wedge \mathbb{R}[\mathtt{c}\ \sigma t_1 \dots \sigma t_{\alpha(\mathtt{c})}]\Uparrow$. We consider a normal-order reduction of $\mathbb{R}[\lambda x.\mathtt{bot}]$.

   $\mathbb{R}[\cdot] \equiv \mathbb{R}'[\mathtt{case}_A\ [\cdot]\ \dots] :$ Then $\mathbb{R}[\lambda x.\mathtt{bot}]\Uparrow$ would hold

   $\mathbb{R}[\cdot] \equiv \mathbb{R}'[[\cdot]\ r] :$ We would obtain $\mathbb{R}[\lambda x.\mathtt{bot}] \xrightarrow{*}_{\text{no}} \mathbb{R}'[\mathtt{bot}]\Uparrow$

   $\mathbb{R}[\cdot] \equiv [\cdot] : \mathbb{R}[\mathtt{c}\ \sigma t_1 \dots \sigma t_{\alpha(\mathtt{c})}] \xrightarrow{*}_{\text{no}} \mathtt{c}\ \sigma t_1 \dots \sigma t_{\alpha(\mathtt{c})}\Downarrow$.

   All cases contradict the assumption.

3. For the context $\mathbb{C}[\cdot] \overset{\text{def}}{=} [\cdot]\ \mathtt{bot}$ we observe $\mathbb{C}[\lambda x_1 \dots x_n.\mathtt{bot}] \to_{\text{no}} \lambda x_2 \dots x_n.\mathtt{bot}$, but $\mathbb{C}[\mathtt{c}\ t_1 \dots t_{\alpha(\mathtt{c})}] = \mathtt{c}\ t_1 \dots t_{\alpha(\mathtt{c})}\ \mathtt{bot} \in \mathit{DIT}$ thus $\lambda x_1 \dots x_n.\mathtt{bot} \not\leq_c \mathtt{c}\ t_1 \dots t_{\alpha(\mathtt{c})}$.

4. This is a direct consequence of proposition 2.96 and the fact that $\mathtt{bot} \leq_c \lambda x.\mathtt{bot}$.

5. For $s\Downarrow_S \mathtt{c}_{A,i}\ s_1 \dots s_{\alpha(\mathtt{c}_{A,i})}$ we define $\mathbb{C}[\cdot] \overset{\text{def}}{=} \mathtt{case}_A\ [\cdot]\ \mathtt{c}_{A,1} \dots \mathtt{c}_{A,|A|}$ and obtain $\mathbb{C}[s]\Downarrow$, but $\mathbb{C}[\lambda x.t]\Uparrow$ so $s \not\leq_c t$.

6. We can see this with the trivial context: $s\Downarrow \wedge t\not\Downarrow$

7. We define $\mathbb{C}[\cdot] \overset{\text{def}}{=} (\lambda y.[\cdot])\ \mathtt{bot}$ then $\mathbb{C}[s] \xrightarrow{*}_{\text{no}} \mathbb{R}[x]\Downarrow$, but $\mathbb{C}[t] \xrightarrow{*}_{\text{no}} \mathbb{R}'[\mathtt{bot}]\Uparrow$. □

With these relations we can draw the diagram of figure 2.3.

Lemma 2.113. *Let $s, t \in \Lambda$ with $s \leq_c t$ then $s\Downarrow_F \wedge s \not\equiv_c \lambda x.\mathtt{bot} \implies t\Downarrow_F$.*

Proof. Assume the statement does not hold. $s\Downarrow_F \wedge s \not\equiv_c \lambda x.\mathtt{bot} \wedge t\not\Downarrow_F$. $t\not\Downarrow_F \iff t\Uparrow \vee t\Downarrow_S \vee t \xrightarrow{*}_{\text{no}} \mathbb{R}[x]$. Definition 2.87 implies $s\Downarrow \implies t\Downarrow$.

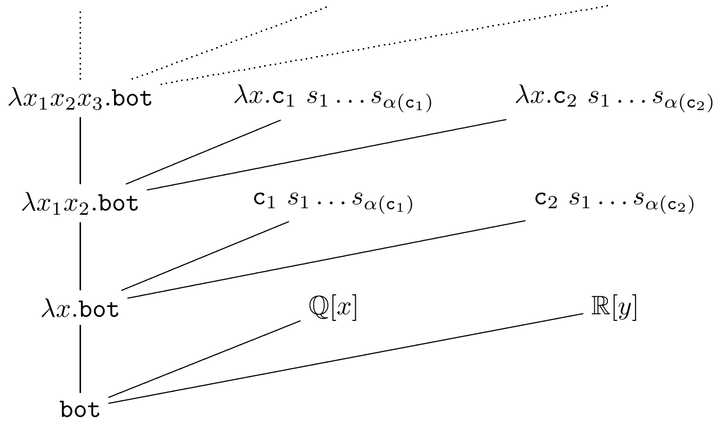

$\lambda x_1 x_2 x_3.\mathtt{bot}$
$\lambda x.\mathtt{c}_1\ s_1 \dots s_{\alpha(\mathtt{c}_1)}$
$\lambda x.\mathtt{c}_2\ s_1 \dots s_{\alpha(\mathtt{c}_2)}$
$\lambda x_1 x_2.\mathtt{bot}$
$\mathtt{c}_1\ s_1 \dots s_{\alpha(\mathtt{c}_1)}$
$\mathtt{c}_2\ s_1 \dots s_{\alpha(\mathtt{c}_2)}$
$\lambda x.\mathtt{bot}$
$\mathbb{Q}[x]$
$\mathbb{R}[y]$
$\mathtt{bot}$

Figure 2.3: Diagram of $\langle \Lambda, \leq_c \rangle$.

1. If $s{\Downarrow}_F \wedge t{\Downarrow}_S$ then for $\mathbb{C}[\cdot] \overset{\text{def}}{=} [\cdot]\ r$ with $r \not\equiv_c \mathtt{bot}$ we find $\mathbb{C}[s] \rightarrow_{\text{no}} s' \not\equiv_c \mathtt{bot}$ so $\mathbb{C}[s]{\Downarrow}$, but $\mathbb{C}[t]{\Uparrow}$ since $\mathbb{C}[t] \in \mathit{DIT}$. This contradicts $s \leq_c t$.
2. If $s{\Downarrow}_F \wedge t \xrightarrow{*}_{\text{no}} \mathbb{R}[x]$ we apply lemma 2.112 (6) to obtain a contradiction to $s \leq_c t$.

Thus the assumption is false and the claim proved. □

2.5.5 Adequacy and full abstraction

Intuitively, adequacy of a semantics means two properties:

1. execution does not change meaning and
2. the result of this execution corresponds exactly to the semantic value of the program.

Together the two properties allow to reason about the operational behavior of a program from semantic observations. For the contextual semantics this means

Definition 2.114 (adequacy). The contextual semantics is *adequate*, iff

$$\forall s, t \in \Lambda : s \rightarrow t \implies s \equiv_c t \qquad (2.9)$$

$$\forall s \in \Lambda : s{\Uparrow} \iff s \equiv_c \mathtt{bot} \qquad (2.10)$$

Theorem 2.115. *The contextual semantics is adequate.*

Proof. Theorem 2.93 proves (2.9) and proposition 2.108 implies (2.10). □

Another important property of semantics is full abstraction with respect to some observation. Intuitively, this property requires that the expressions differentiated by the operational observation are exactly the ones differentiated by the semantics. Since the semantic equivalence for contextual semantics is defined to be precisely the operational observation of termination after insertion into a context, this semantics is naturally always fully abstract with respect to this observation.

2.5.6 Chains

Definition 2.116 ($\bigsqcup$, lub). Let $s_1 \leq_c s_2 \leq_c \dots$ be an ascending chain of Λ-expressions. $s \in \Lambda$ is the *least upper bound* or *lub* of the s_i, $s \equiv_c \bigsqcup_i s_i$, iff $\forall i : s_i \leq_c s$ and $\forall i : s_i \leq_c t \implies s \leq_c t$.

Definition 2.117 ($\bigsqcup^c$). Let $s_1 \leq_c s_2 \leq_c \dots$ be a chain of Λ-expressions. $s \in \Lambda$ is the *contextually least upper bound*, pronounced *club*, of the s_i, written $s \equiv_c \bigsqcup_i^c s_i$, iff

$$\forall \mathbb{C}[\cdot] : \bigsqcup_i \mathbb{C}[s_i] \equiv_c \mathbb{C}[s] \tag{2.11}$$

We also define $\bigsqcup^c$ as an operator on sets

$$\bigsqcup^c M \overset{\text{def}}{=} \{s \in \Lambda | s \equiv_c \bigsqcup_i^c s_i \wedge s_i \leq_c s_{i+1} \wedge s_i \in M\}.$$

It is unknown whether the condition 2.11 is satisfied for every lub or not. The investigation of this question is outside the scope of this work. The property is needed in the proof of continuity of contexts. We do not have a counter-example, i.e. we do not know of a context $\mathbb{C}[\cdot]$ and two ascending chains $s_i \leq_c s_{i+1}$ and $t_i \leq_c t_{i+1}$ having the same lub s, but for which $\bigsqcup_i \mathbb{C}[s_i] \not\equiv_c \bigsqcup_i \mathbb{C}[t_i]$ holds.

Lemma 2.118. *$\bigsqcup^c$ is monotone on sets of Λ-expressions.*

Proof. If $M \subseteq N$ then any chain from M is contained in N thus $\bigsqcup^c M \subseteq \bigsqcup^c N$. □

Remark 2.119. $\texttt{bot} \equiv_c \bigsqcup_i^c s_i \iff \forall i : s_i \equiv_c \texttt{bot}$

Proof. Follows immediately from the definition of club with the empty context. □

The club is unique up to $\equiv_c$.

Lemma 2.120. *Let $s_1 \leq_c s_2 \leq_c \ldots, \forall i : s_i \in \Lambda, s, t \in \Lambda$ and $s \equiv_c \bigsqcup_i^c s_i$. Then*

$$t \equiv_c \bigsqcup_i^c s_i \iff t \equiv_c s.$$

Proof. Both implications follow directly from transitivity of $\equiv_c$. □

Lemma 2.121. *Let $s_1 \leq_c s_2 \leq_c \ldots$ and $I \subseteq \mathbb{N}$ with $|I| = \infty$, then $\bigsqcup_i^c s_i$ exists iff $\bigsqcup_{i \in I}^c s_i$ exists and $\bigsqcup_i^c s_i \equiv_c \bigsqcup_{i \in I}^c s_i$.*

Proof. Since I is infinite $\bigsqcup_{i \in I} s_i \equiv_c \bigsqcup_i s_i$. We obtain $\bigsqcup_i^c s_i \equiv_c \bigsqcup_i s_i \equiv_c \bigsqcup_{i \in I} s_i \equiv_c \bigsqcup_{i \in I}^c s_i$. The last equivalence is due to the fact that $\bigsqcup_i s_i$ already is a club and the club is unique up to $\equiv_c$. □

The club of a constant chain is this constant value.

Lemma 2.122. *Let $s, s_i \in \Lambda$ with $s_i \equiv_c s_{i+1}$ and $s \equiv_c \bigsqcup_i^c s_i$. Then $s \equiv_c s_i$ for any i.*

Proof. Every chain element is the lub and our premise implies that the lub is a club, so the statement is proved. □

Definition 2.123 (continuity). A context $\mathbb{C}[\cdot]$ is called *continuous* iff for all chains $s_1 \leq_c s_2 \leq_c \ldots$ having a club $s \equiv_c \bigsqcup_i^c s_i$ there is $\bigsqcup_i^c \mathbb{C}[s_i] \in \Lambda$ and $\mathbb{C}[s] \equiv_c \bigsqcup_i^c \mathbb{C}[s_i]$.

Theorem 2.124 (Continuity of contexts). *Every context is continuous.*

Proof. We show that for any context $\mathbb{C}[\cdot]$ and ascending chain $s_i \leq_c s_{i+1}$ satisfying $s \equiv_c \bigsqcup_i^c s_i$ also $\mathbb{C}[s] \in \Lambda$ and $\mathbb{C}[s] \equiv_c \bigsqcup_i^c \mathbb{C}[s_i]$ is satisfied.

1. $\mathbb{C}[s] \in \Lambda$ trivially holds.
2. From $s \equiv_c \bigsqcup_i^c s_i$ we deduce $\forall \mathbb{D}[\cdot] : \mathbb{D}[s] \equiv_c \bigsqcup_i \mathbb{D}[s_i]$. This also holds for $\mathbb{D}[\cdot] \equiv \mathbb{C}[\cdot]$.

3. Likewise, $\forall \mathbb{E} : \mathbb{E}[\mathbb{C}[s]] \equiv_c \bigsqcup_i \mathbb{E}[\mathbb{C}[s_i]]$.

2 and 3 together imply $\mathbb{C}[s] \equiv_c \bigsqcup_i^c \mathbb{C}[s_i]$. □

There is also a kind of continuity for multi-hole contexts. We consider contexts with two holes only since the argumentation straightforwardly generalizes to more than two holes. These observations will later allow us to lift the club out from contexts in which there are multiple independent clubs and to consider only the chain elements with identical indices together. In this sense we are cutting diagonally through the chains.

Theorem 2.125. *Let $s_1 \leq_c s_2 \leq_c \dots$ and $t_1 \leq_c t_2 \leq_c \dots$ be ascending chains, then*

$$\forall \mathbb{C}[\cdot_1, \cdot_2] : \bigsqcup_i^c \mathbb{C}[s_i, t_i] \equiv_c \mathbb{C}[\bigsqcup_i^c x_i, \bigsqcup_i^c y_i].$$

Proof. The proof is in three parts.

upper bound: Defining $\mathbb{D}_i[\cdot] \overset{\text{def}}{=} \mathbb{C}[s_i, \cdot]$ it is easy to see that $\forall i : \mathbb{C}[s_i, t_i] \equiv \mathbb{D}_i[t_i] \leq_c \mathbb{D}_i[\bigsqcup_i^c t_i] \equiv \mathbb{C}[s_i, \bigsqcup_i^c t_i]$ and with $\mathbb{D}'[\cdot] = \mathbb{C}[\cdot, \bigsqcup_i^c t_i]$ we see $\forall i : \mathbb{C}[s_i, \bigsqcup_i^c t_i] \equiv \mathbb{D}'[s_i] \leq_c \mathbb{D}'[\bigsqcup_i^c s_i] \equiv \mathbb{C}[\bigsqcup_i^c s_i, \bigsqcup_i^c t_i]$.

least: Let t be an upper bound on $\mathbb{C}[s_i, t_i]$. For every fixed j it must be that $\forall i : \mathbb{C}[s_i, t_j] \leq_c t$, since with $k = \max\{i, j\}$ we have $\mathbb{C}[s_i, t_j] \leq_c \mathbb{C}[s_k, t_k] \leq_c t$. Consequently, $\forall j : \mathbb{C}[\bigsqcup_i^c s_i, t_j] \leq_c t$ and thus $\mathbb{C}[\bigsqcup_i^c s_i, \bigsqcup_i^c t_i] \leq_c t$.

$\forall \mathbb{C}[\cdot_1, \cdot_2], \mathbb{D}[\cdot] : \bigsqcup_i \mathbb{D}[\mathbb{C}[s_i, t_i]] \equiv_c \mathbb{D}[\mathbb{C}[\bigsqcup_i^c s_i, \bigsqcup_i^c t_i]]$. We can combine the two contexts and obtain the equivalent statement $\forall \mathbb{C}[\cdot_1, \cdot_2] : \bigsqcup_i \mathbb{C}[s_i, t_i] \equiv_c \mathbb{C}[\bigsqcup_i^c s_i, \bigsqcup_i^c t_i]$. But this is exactly what we have already shown for being a least upper bound in the cases above. □

Corollary 2.126. *Let $s_1 \leq_c s_2 \leq_c \dots$ and $t_1 \leq_c t_2 \leq_c \dots$ be ascending chains with $s \equiv_c \bigsqcup_i^c s_i$ and $t \equiv_c \bigsqcup_i^c t_i$. Then*

$$f \bigsqcup_i^c s_i \bigsqcup_i^c t_i \equiv_c \bigsqcup_i^c f\ s_i\ t_i \tag{2.12}$$

$$(\bigsqcup_i^c s_i) \bigsqcup_i^c t_i \equiv_c \bigsqcup_i^c s_i\ t_i \tag{2.13}$$

2.5.7 Fixpoint combinators

We will now present a syntactic criterion which will allow to state the minimal number of normal-order reductions necessary for a position which differs in two expressions to become visible in a reduction context. First we define a dual notion to the notion of reduction contexts.

Definition 2.127 (argument context). An *argument context* is defined by the following grammar:

$$\begin{array}{rcl} \mathbb{A}[\cdot] & \rightarrow & e\ [\cdot], e \not\equiv \texttt{case}_A \\ & | & \texttt{case}_A\ e\ e_1 \dots e_{i-1}\ [\cdot]\ e_{i+1} \dots e_{|A|} \\ & | & \lambda x.[\cdot] \end{array}$$

If not otherwise specified $\mathbb{A}[\cdot], \mathbb{Y}[\cdot]$ and $\mathbb{Z}[\cdot]$ will range over argument contexts.

Lemma 2.128. *Every context $\mathbb{C}[\cdot]$ can be represented as the composition of reduction contexts $\mathbb{R}_1[\cdot], \dots, \mathbb{R}_n[\cdot]$ and argument contexts $\mathbb{A}_1[\cdot], \dots, \mathbb{A}_m[\cdot]$ with $n > 0, m \geq 0$.*

Proof. By induction on the structure of $\mathbb{C}[\cdot]$. □

Definition 2.129. Two Λ-expressions s, t *differ at argument depth* n, if there is a position ρ in s and t at which their labels differ and the contexts corresponding to position ρ have exactly n argument contexts. Thus

$n = 0: \; s(\rho) \neq t(\rho) \implies \exists \mathbb{R}[\cdot], \mathbb{S}[\cdot], u, v \in \Lambda : \mathbb{R}[\cdot]_\rho \equiv \mathbb{S}[\cdot]_\rho \equiv [\cdot] \wedge s \equiv \mathbb{R}[u] \wedge t \equiv \mathbb{S}[v]$.

$n > 0: \; s(\rho) \neq t(\rho) \implies \exists \mathbb{R}[\cdot], \mathbb{S}[\cdot], \mathbb{A}[\cdot], \mathbb{Z}[\cdot], u, v \in \Lambda : \mathbb{R}[\mathbb{A}[\cdot]]_\rho \equiv \mathbb{S}[\mathbb{Z}[\cdot]]_\rho \equiv [\cdot] \wedge s \equiv \mathbb{R}[\mathbb{A}[u]] \wedge t \equiv \mathbb{S}[\mathbb{Z}[v]] \wedge u$ and v differ at argument depth $n-1$.

Accordingly, we say that s and t *are equal above argument depth* n, if they do not differ for any argument depth $m < n$.

The argument depth has the desired property of requiring expressions that are equal above argument depth n to be normal-order reduced at least n times before expressions result that differ in their respective reduction contexts. The base case for the induction can be seen directly from definition 2.127. The induction step is established in the following

Lemma 2.130. *Let $s, t \in \Lambda$ be equal above argument depth $n > 0$ and let $s \rightarrow_{no} s', t \rightarrow_{no} t'$, then s' and t' are equal above argument depth $n-1$.*

Proof. If the normal-order redexes of s and t, u and v, differ at all, then they do not do so above argument depth n, because every prefix of a corresponding reduction context is itself a reduction context (and not an argument context). Since $n > 0$ either $u \equiv (\lambda x.u_1)\, u_2$ and $v \equiv (\lambda x.v_1)\, v_2$, u_1 and v_1 are equal above argument depth $n-1$, and u_2 and v_2 are equal above argument depth $n-1$, or $u \equiv \texttt{case}_A\; a\; u_1 \ldots u_{|A|}$ and $v \equiv \texttt{case}_A\; b\; v_1 \ldots v_{|A|}$, a and b

are equal above argument depth n, and u_i and v_i are equal above argument depth $n-1$.

Case 1 : If the $\rightarrow_B$-reduction substitutes a free occurrence of x below an argument context then on the path to differently labeled positions (if they exist) there are at least n argument contexts. If it substitutes x in a reduction context then there are still $n-1$ on the path to a possibly existing, differently labeled position.

Case 2 : $a \equiv \texttt{c}_{A,i}\; a_1 \ldots a_{\alpha(\texttt{c}_{A,i})}$ and $b \equiv \texttt{c}_{A,i}\; b_1 \ldots b_{\alpha(\texttt{c}_{A,i})}$ and the a_i and the b_i are equal above argument depth $n-1$. Thus $u_i\; a_1 \ldots a_{\alpha(\texttt{c}_{A,i})}$ and $v_i\; b_1 \ldots b_{\alpha(\texttt{c}_{A,i})}$ are equal above argument depth $n-1$.

It follows that s' and t' are equal above argument depth $n-1$. □

Corollary 2.131. *If two expressions are equal above argument depth $n > 0$ then the position of their normal-order redex is the same if it exists and it is reduced by the same alternative of the $\rightarrow_B$-reduction.*

Proof. Normal-order redexes are found in reduction contexts, i.e. not below an argument context. Since the reduction context of a normal-order redex is not maximal, the alternative with which it is $\rightarrow_B$-reduced is uniquely determined, because either there is only one maximal reduction context and there is an abstraction in its hole or there are two maximal reduction contexts $\mathbb{R}^1[\cdot]$ and $\mathbb{R}^2[\cdot]$, such that the expression reduced can be written $\mathbb{R}[\mathbb{R}^1[s_1]]$ where $s_1 \equiv \texttt{case}_A$ or $\mathbb{R}[\texttt{case}_A\; \mathbb{R}^2[s_2]]$. Even in the latter two cases though s_1 and s_2 will not be below an argument context so that the corresponding positions will be equal above argument depth n. It follows that the normal-order redexes $\to_B$-reduce with the same alternative. □

We will now approach the presentation of two important properties, one is that $\boldsymbol{\theta}$ is a fixpoint combinator, the other is that the fixpoint $\boldsymbol{\theta}\ t$ is approximated by the iterations of t and is equivalent to their club.
In the proof of theorem 2.133 below we employ the following characterization of the club of an ascending chain.

Lemma 2.132. *Let $s_1 \leq_c s_2 \leq_c \dots$ be an ascending chain and let $s \in \Lambda$. If*

$$\forall i : s_i \leq_c s, \text{ and} \tag{2.14}$$
$$\forall \mathbb{C}[\cdot] : \mathbb{C}[s]\Downarrow \implies \exists i : \mathbb{C}[s_i]\Downarrow \tag{2.15}$$

then

$$s \equiv_c \bigsqcup\nolimits_i^c s_i.$$

Proof. We show $\forall \mathbb{D}[\cdot] : \mathbb{D}[s] \equiv_c \bigsqcup_i^c \mathbb{D}[s_i]$. From (2.14) and since $\leq_c$ is a precongruence $\bigsqcup_i^c \mathbb{D}[s_i] \leq_c \mathbb{D}[s]$.
Let $\mathbb{D}[\cdot]$ be an arbitrary context and t a Λ-expression which is an upper bound for the $\mathbb{D}[s_i]$, i.e. for which $\forall i : \mathbb{D}[s_i] \leq_c t$. If for some context $\mathbb{C}[\cdot] : \mathbb{C}[\mathbb{D}[s]]\Downarrow$ then with (2.15) we get $\exists i : \mathbb{C}[\mathbb{D}[s_i]]\Downarrow$ and thus $\mathbb{C}[t]\Downarrow$, so now we have $\mathbb{C}[\mathbb{D}[s]]\Downarrow \implies \mathbb{C}[t]\Downarrow$ for all $\mathbb{C}[\cdot]$ which is equivalent to $\mathbb{D}[s] \leq_c t$. Hence $\mathbb{D}[s]$ is the least upper bound, so $\mathbb{D}[s] \leq_c \bigsqcup_i^c \mathbb{D}[s_i]$. □

Theorem 2.133. *Let $t \in \Lambda$, then*

$$\boldsymbol{\theta}\ t \equiv_c \bigsqcup\nolimits_i^c t^i\ \texttt{bot}.$$

Proof.

$t^i\ \texttt{bot} \leq_c t^{i+1}\ \texttt{bot}$: $\texttt{bot} \leq_c t\ \texttt{bot}$, the rest follows from monotonicity of contexts.

Criterion of lemma 2.132:

$t^i\ \texttt{bot} \leq_c \boldsymbol{\theta}\ t$: $\texttt{bot} \leq_c \boldsymbol{\theta}\ t$ and since $\leq_c$ is a precongruence $t^i\ \texttt{bot} \leq_c \boldsymbol{\theta}\ t$ implies $t^{i+1}\ \texttt{bot} \leq_c t\ (\boldsymbol{\theta}\ t)$. Since $\boldsymbol{\theta}\ t \xrightarrow{*} t\ (\boldsymbol{\theta}\ t)$ we apply theorem 2.93 to conclude that $t^{i+1}\ \texttt{bot} \leq_c t\ (\boldsymbol{\theta}\ t)$ implies $t^{i+1}\ \texttt{bot} \leq_c \boldsymbol{\theta}\ t$ and by induction we obtain $\forall i : t^i\ \texttt{bot} \leq_c \boldsymbol{\theta}\ t$.

$\mathbb{C}[\boldsymbol{\theta}\ t]\Downarrow \implies \exists i : \mathbb{C}[t^i\ \texttt{bot}]\Downarrow$: Let $\mathbb{C}[\cdot]$ be an arbitrary context with $\mathbb{C}[\boldsymbol{\theta}\ t]\Downarrow$ and let i be the number of normal-order reductions from $\mathbb{C}[\boldsymbol{\theta}\ t]$ to WHNF. $\mathbb{C}[t^{i+1}(\boldsymbol{\theta}\ t)]$ has no more than i normal-order reductions to a WHNF. This follows from $\boldsymbol{\theta}\ t \xrightarrow{*} t^{i+1}\ (\boldsymbol{\theta}\ t)$ and theorem 2.51. The expressions $\mathbb{C}[t^{i+1}\ (\boldsymbol{\theta}\ t)]$ and $\mathbb{C}[t^{i+1}\ \texttt{bot}]$ are equal above argument depth $i+1$. Hence the $\to_{\text{no}}$-reduction sequence to WHNF from $\mathbb{C}[t^{i+1}\ (\boldsymbol{\theta}\ t)]$ can also be applied to $\mathbb{C}[t^{i+1}\ \texttt{bot}]$. Let s' and t' respectively be the results of these reductions. s' and t' are equal above argument position 1, and in particular t' is a WHNF. □

Theorem 2.134. *$\boldsymbol{\theta}\ t$ is the smallest fixpoint of t.*

Proof. $\boldsymbol{\theta}\ t \equiv (\lambda z.\lambda x.x\ (z\ z\ x))\ \texttt{A}\ t \xrightarrow{2}_{\text{no}} t\ (\texttt{A}\ \texttt{A}\ t) \equiv t\ (\boldsymbol{\theta}\ t)$ and thus $\boldsymbol{\theta}\ t$ is a fixpoint of t.
To show that this is the *least* such fixpoint requires to show $t\ f \equiv_c f \implies \boldsymbol{\theta}\ t \leq_c f$. Assume the premise, then for all $i : t^i\ \texttt{bot} \leq_c f$ since $\texttt{bot} \leq_c f$ and from $t\ (t^i\ \texttt{bot}) \leq_c t\ f$ we may conclude $t^{i+1}\ \texttt{bot} \leq_c f$. Since by theorem 2.133 above $\boldsymbol{\theta}\ t$ is the least upper bound of the $t^i\ \texttt{bot}$ we obtain $\boldsymbol{\theta}\ t \leq_c f$. □

2.5.8 Super-combinators

We have defined Λ as a λ-calculus extended with constructors and case. One could argue that there is quite a difference between the language of our analysis and the core languages used in the implementations of functional languages.

In this section we narrow this gap by showing how to integrate the use and definition of recursive super-combinators into our language. For the fundamental considerations in this chapter recursive super-combinators would have been rather obstructive, e.g. the notion of context would have to be changed to consist of a set of super-combinator definitions and an expression where the position of the hole could be inside either of the two.

We will present the addition of recursive super-combinators as a notational convention by translating them to Λ. We will only describe how this is done in principle and not give much detail.

As we have already seen $\boldsymbol{\theta}$ is a fixpoint combinator computing the least fixed point $\boldsymbol{\theta}\ s$ for any Λ-expression s. In order to translate a recursive n-ary super-combinator into a Λ-expression we start by wrapping the right hand side, e, of the super-combinator's definition in n abstractions yielding $\lambda x_1 \ldots x_n.e$. If e was the right hand side of a definition for super-combinator H and e uses only the super-combinator name H then it will be sufficient to abstract from H in e, i.e. to form $\lambda f x_1 \ldots x_n.e[{}^f/_H]$ and to use the fixpoint of this expression as the meaning of the super-combinator H.

Since Λ allows the use of constructors this principle is easily applied to super-combinator definitions referencing more than one super-combinator, possibly mutually recursively. We introduce a type with a 0-ary constructor $\mathtt{c}_H$ for each super-combinator H of the program and replace any occurrence of a super-combinator name S in the body of a super-combinator definition with $(F\ \mathtt{c}_S)$.

Let e_S be the result of this replacement wrapped in enough abstractions to account for the arity of S. We form the Λ-expression $P \overset{\text{def}}{=} \lambda F.\lambda x.\mathtt{case}_A\ x\ e_{S_1} \ldots e_{S_n}$. The functionality of super-combinator S can then be selected with the Λ-expression $(\boldsymbol{\theta}\ P)\ \mathtt{c}_S$ and we consider this to be the meaning of S.

2.5.9 Approximation

We have characterized the club-operation by demonstrating its club-closure for primitive sets. A further question to characterize the club-operation is to ask for a set of expressions containing the clubs for arbitrary ascending chains of Λ-expressions. A suggesting candidate for such a set would be Λ itself, i.e. "Is there an ascending chain of Λ-expressions which cannot have a club in Λ?" We will answer this question affirmatively by providing an ascending chain whose club would be a non-computable function, if it would exist.

Fact 2.135. *There are uncountably many infinite strings over $\{0,1\}$, but only countably many Λ-expressions. Hence there must be infinite strings over $\{0,1\}$ for which there cannot be a Λ-expression.*

We obtain the

Proposition 2.136. *There are ascending chains of Λ-expressions that cannot have a club in Λ.*

Proof. Let $a_1 a_2 a_3 \ldots$ be any infinite string over $\{0,1\}$. Its finite prefixes can be encoded as finite lists of elements from $\{\mathtt{0},\mathtt{1}\}$ which end in $\mathtt{bot}$. We define Λ-expressions $s_i \overset{\text{def}}{=} a_1 : \ldots : a_i : \mathtt{bot}$ obviously forming an ascending chain. If a Λ-expression would be the club of this chain then it would need to be equivalent to the infinite list $a_1 : \ldots : a_i : \ldots$. This follows from lemma 2.110. Thus for some of the infinite strings $a_1 a_2 a_3 \ldots$ there cannot be such a Λ-expression. $\square$

In a large portion of the literature on semantics of programming languages e.g. [Bur91, Mit96] CPOs are used as the semantic domains for the syntactic language constructs. CPOs can be defined as in e.g. [DP90]:

Definition 2.137. $(P, \leq)$ is a *CPO*, iff for all directed subsets $D \subseteq P$ there is $\bigsqcup D$. D is *directed*, if for all finite subsets $E \subseteq D$ there is a $z \in D$ with $z \in E^u$ where E^u is the set of largest elements with respect to $\leq$.

Since the elements of an ascending chain trivially form a directed set, proposition 2.136 implies

Proposition 2.138. $(\Lambda, \leq_c)$ *is not a CPO.*

3 Demands

In this chapter we define *demands* which are expressions representing subsets of Λ. These sets have the property of being closed with respect to $\equiv_c$-equivalence. Demand representations are provided for diverging expressions, for expressions having an FWHNF, and for expressions having a saturated constructor WHNF. A high degree of detail is available for the latter, since every constructor from the program has a corresponding demand constructor. An inductive construction as well as union and intersection of the represented sets are also provided. Λ-expressions which can be evaluated to arbitrary depth enter by taking least upper bounds of ascending chains of elements evaluating to finite depth. Not every lub of an ascending chain is considered for the semantics of demands. To ensure continuity of context we allow only those that satisfy an additional restriction.
We start by presenting the syntax of demands in section 3.1, then we define the Λ-expressions represented by a demand expression, followed by the concretization which captures our notion of the "meaning" of a demand expression.

3.1 Syntax

Definition 3.1. Λ_C is the set of demand expressions, which is generated by the grammar in figure 3.1 with start symbol E'.
The set of closed demand expressions is Λ_C^0. We also write Λ_C^p for

demand expr	$E' \to W \mid S \mid U' \mid C$	
	$\mid K' \mid N \mid V'$	
where expr	$W \to E'$ `where` $T = N$	
intersection	$S \to E'_1 \cap \cdots \cap E'_n$	$n \geq 0$
union	$U' \to \langle E'_1, \ldots, E'_n \rangle$	$n \geq 0$
demand constant	$C \to$ `Bot` $\mid$ `Fun`	
demand name	$N \to \langle\text{demandid}\rangle$	
demand constructor	$K' \to \mathtt{c}_{A,i}\ E'_1 \ldots E'_{\alpha(\mathtt{c}_{A,i})}$	$1 \leq i \leq \lvert A \rvert$
demand pattern	$T \to \mathtt{c}_{A,i}\ T_1 \ldots T_{\alpha(\mathtt{c}_{A,i})}$	$1 \leq i \leq \lvert A \rvert$
	$\mid$ `Bot` $\mid V'$	
demand variable	$V' \to \langle\text{varid}\rangle$	
demand definition	$D' \to N = E'$	
primitive demands	$S' \to \mathtt{c}_{A,i}\ S'_1 \ldots S'_{\alpha(\mathtt{c}_{A,i})}$	
	$\mid$ `Bot` $\mid$ `Fun`	

Figure 3.1: Syntax of Λ_C

the set of primitive demands.

Variables in patterns need not be unique.

Notation 3.2. It will often be necessary to speak of specific parts of demand expressions, therefore we name some of these parts: the demand term to the left of a `where` is the *contribution* of a `where`-expression, the equation to its right is the *match* consisting of the *pattern* and the *name* of the `where`-expression. The direct subexpressions of unions, intersections and constructors are their *components*, for constructors we will also refer to them as *arguments*. Unless otherwise stated we will assume all demand expressions to be closed except the pattern and contribution of `where`-expressions and the notion of substitution is straightforwardly extended to open demand expressions.

3.1.1 Jocs

Our calculi can handle inputs in which more than one variable occurs free. In a solution for all these variables the demands for each of the constrained variables are typically not independent. As a means for representing the interdependencies among the variables we use constructors introduced for this sole purpose. We call such constructors *joint-constructors* or *jocs* since they make it possible to *jointly* constrain variables of the input. We assume that jocs of every required arity are defined and that they do not occur anywhere in the input program. Consequently, no expression in the program will evaluate to an SCWHNF using a joc as its top-level constructor unless purposefully introduced and this is the only feature distinguishing jocs from constructors in general. Our notation for jocs are brackets, i.e. $[\dots]$. Since we will never take an interest in the joc by itself, but only in its components, their number and the fact that they are the arguments of a joc we will also call expressions with a top-level joc (in Λ as well as in Λ_C) jocs.

Definition 3.3 (joc). An n-ary *joc* is an n-ary constructor $[]_n \in \Lambda$ (and also $[]_n \in \Lambda_C$) which we write in mixfix notation with its arguments $a_1, \dots, a_n$, i.e $[a_1, \dots, a_n]_n$. Since its arity will be

apparent most of the time we will just write $[a_1, \dots, a_n]$.

Example 3.4. Consider, for example, the jocs $[x, y] \in \Lambda$ or $[\texttt{Bot}, \texttt{[]}] \in \Lambda_C$.

It will sometimes be convenient to identify singleton jocs with their argument. This will be the only exception we make from representing every n-ary joc exactly as an n-ary constructor.
Jocs frequently occur in relation to `where`-expressions, where they have the effect of enforcing a simultaneous binding of the pattern variables. In that context we will also write $\vec{x}$ for the joc $[x_1, \dots, x_n]$.

3.2 Semantics

Demands are a representation of particular subsets of Λ. They have the property that equivalent Λ-expressions belong to the same demand. Their semantics is given by a two-stage process: the first stage is inductive and in the second the set obtained from the first is closed with respect to contextual least upper bounds of ascending chains.
Demands are a very powerful tool in more than one respect. We will show that the demand language's expressive power equals that of Turing-machines. Furthermore, we will show demands exist having a fixpoint of a function as their semantics, but for which the semantics can neither be the least nor the greatest fixpoint of a function.

3.2.1 Representation

The representation of a demand is the first stage in defining the semantics of demands. It is obtained as the fixpoint of approximations and defines a set of expressions evaluating to finite depth associated with a demand.

Example 3.5. To motivate the development ahead in this section we present some examples for the representation, $\eta(\cdot)$, of demands. We assume the demands `Inf1` and `Peano` are defined as $\texttt{Inf1} = \langle \texttt{Bot}, \texttt{1 : Inf1}\rangle$ and $\texttt{Peano} = \langle \texttt{Zero}, \texttt{Succ Peano}\rangle$.

$$\begin{aligned}
\eta(\texttt{Bot}) &= \{t | t\Uparrow\} \\
\eta(\langle \texttt{True}, \texttt{False}\rangle) &= \{t | t\Downarrow_S \texttt{True} \vee t\Downarrow_S \texttt{False}\} \\
\eta(\langle\rangle) &= \emptyset \\
\eta(\texttt{Peano}) &= \{t | \exists i \in \mathbb{N}_0 : t\Downarrow_S \texttt{Succ}^i\ \texttt{Zero}\} \\
\eta(\texttt{Inf1}) &= \{t | \exists i \in \mathbb{N}_0 : t \xrightarrow{*} (\texttt{1:})^i\ r \wedge r\Uparrow\}.
\end{aligned}$$

We want to infer the existence of the fixed point using the Theorem of Knaster and Tarski, therefore we need to show that the approximations are total mappings in a complete lattice and thus form a complete lattice themselves and that the operation which improves approximations is monotone.

Definition 3.6. Let $\mathcal{C}$ be a set of demand definitions. We define the set of *demand names* occurring in $\mathcal{C}$ as $N(\mathcal{C}) = \{N | N = C \in \mathcal{C}\}$.

Definition 3.7. Let $f, g : B \to A$ where $(A, \leq)$ is an ordered set and B an arbitrary set. The *point-wise order* on $B \to A$ is then $f \leq g \iff \forall x \in B : (f(x) \leq g(x))$.

Lemma 3.8. *$\mathcal{P}(\Lambda)^{N(\mathcal{C})}$ is a complete lattice with the point-wise order.*

Proof. Let M be any set, $(\mathcal{P}(M), \subseteq)$ is a complete lattice, hence $(\mathcal{P}(\Lambda), \subseteq)$ is a complete lattice. Furthermore the mapping from any set into a complete lattice is a complete lattice with the point-wise order.[DP90] □

0 denotes the function that maps every name to the empty set. We will overload **0** to have this meaning in the complete lattice $\mathcal{P}(M)^{N(\mathcal{C})}$ for any M.
In case only `Bot`, `Fun` and constructors are present in the demand expression, i.e. if the demand is *primitive* (cf. figure 3.1), we can give a direct definition of the primitive terms represented.

Definition 3.9 (primitive terms). Let $\Lambda_p : \Lambda_C^p \to \Lambda$ be the function for which the following holds:

$$\Lambda_p(\texttt{Bot}) \overset{\text{def}}{=} \{t | t\Uparrow\} \tag{3.1}$$

$$\Lambda_p(\texttt{Fun}) \overset{\text{def}}{=} \{t | t\Downarrow_F\} \tag{3.2}$$

$$\Lambda_p(\texttt{c}\ C_1 \dots C_{\alpha(\texttt{c})}) \overset{\text{def}}{=} \{t | t\Downarrow_S \texttt{c}\ t_1 \dots t_{\alpha(\texttt{c})} \wedge t_i \in \Lambda_p(C_i)\} \tag{3.3}$$

The image of Λ_C^p under $\Lambda_p(\cdot)$ is called the set of *primitive expressions*. Λ-expressions which are not primitive will be called *non-primitive* expressions. Similarly, *non-primitive demands* are those that are not primitive.

Example 3.10.

$$\begin{aligned}
\texttt{bot} &\in \Lambda_p(\texttt{Bot}) \\
\texttt{bot bot} &\in \Lambda_p(\texttt{Bot}) \\
\lambda x.\texttt{bot} &\in \Lambda_p(\texttt{Fun}) \\
\texttt{c}\ (\lambda x.\texttt{bot})\ \texttt{1} &\in \Lambda_p(\texttt{c Fun 1})
\end{aligned}$$

Sometimes we will want to convert Λ-expressions to demands such that all $\equiv_c$-equivalent elements are represented. This can be done for some Λ-expressions with the following operation.

Definition 3.11 ($\dot{t}$). Let t be a Λ-expression, then $\dot{t}$ is the demand defined as:

$$\begin{aligned}
\dot{\texttt{bot}} &\overset{\text{def}}{=} \texttt{Bot}\\
\dot{f} &\overset{\text{def}}{=} \texttt{Fun},\ \text{if } f \text{ is an FWHNF.}\\
\dot{t} &\overset{\text{def}}{=} \texttt{c}\ \dot{t_1} \ldots \dot{t}_{\alpha(\texttt{c})},\ \text{if } t \equiv \texttt{c}\ t_1 \ldots t_{\alpha(\texttt{c})} \wedge \texttt{c} \in K\\
\dot{t} &\overset{\text{def}}{=} t,\ \text{if } t \in V \cap V'\\
\dot{t} &\overset{\text{def}}{=} \text{undefined, otherwise.}
\end{aligned}$$

If $\sigma = \{x \mapsto t, \ldots\}$ substitutes Λ-expressions then $\dot{\sigma}$ is defined to be $\{\dot{x} \mapsto \dot{t}, \ldots\}$.

This operation will not transform arbitrary Λ-expressions into demands, e.g. abstractions will not be transformed into demands.

Example 3.12. Let $t \equiv (\lambda x.\texttt{c}\ \underbrace{\texttt{bot} \ldots \texttt{bot}}_{\alpha(c)-1}\ x)\ \texttt{bot}$, then $\dot{t}$ is undefined, but for $t \equiv \texttt{c}\ \underbrace{\texttt{bot} \ldots \texttt{bot}}_{\alpha(c)}$, $\dot{t} \equiv \texttt{c}\ \underbrace{\texttt{Bot} \ldots \texttt{Bot}}_{\alpha(c)}$.

However, the operation is sufficient for transforming into demands those Λ-expressions which are well-suited for representing the equivalence classes with respect to $\equiv_c$.

Lemma 3.13. *If $t \in \Lambda$ consists of constructors, FWHNFs and* `bot` *only, then $\dot{t}$ is a closed demand expression and $t \in \Lambda_p(\dot{t})$.*

Proof. $\dot{\texttt{bot}} \equiv \texttt{Bot}$ and the statement holds. If f is an FWHNF, the statement is satisfied for $\dot{f} \equiv \texttt{Fun}$. If we have an SCWHNF $t \equiv \texttt{c}\ t_1 \ldots t_{\alpha(\texttt{c})}$, then $\dot{t} \equiv \texttt{c}\ \dot{t_1} \ldots \dot{t}_{\alpha(\texttt{c})}$. The t_i consist of constructors, FWHNFs and `bot` only, such that by induction hypothesis the $\dot{t_i}$ are closed demand expressions and $t_i \in \Lambda_p(\dot{t_i})$. Then $\dot{t}$ is also a closed demand expression and $t \in \Lambda_p(\dot{t})$. □

Lemma 3.14. *If $t \in \Lambda$ consists of constructors, variables and* `bot` *only, then $\dot{t}$ is a demand pattern.*

Proof. Analogous to the proof of lemma 3.13. □

Lemma 3.15. *If $t \in \Lambda$ consists only of constructors, variables, FWHNFs and* `bot` *and if θt is primitive, then $\exists \rho : \theta t \in \Lambda_p(\rho\dot{t})$.*

Proof. Let $\{x_1, \ldots, x_n\}$ be the free variables of t. θx_i is primitive and so by lemma 3.25 there is some E_i satisfying $\theta x_i \in \Lambda_p(E_i)$. We define $\rho \overset{\text{def}}{=} \{x_i \mapsto E_i | 1 \leq i \leq n\}$. It is easily seen by induction on the structure of t that $\theta t \in \Lambda_p(\rho\dot{t})$. □

Lemma 3.16. *Let $s \in \Lambda_p(D)$ for some D where $s \equiv_c \bigsqcup_i^c s_i$. Then all but finitely many $s_i \in \Lambda_p(D)$.*

Proof. We distinguish three cases for D.

1. $D \equiv \texttt{Bot}$: $s \leq_c \texttt{bot}$ and thus all $s_i \leq_c \texttt{bot}$.

2. $D \equiv \texttt{Fun}$: There must be an $i_0 : s_{i_0}$ has an FWHNF, otherwise all $s_i \equiv_c \texttt{bot}$ and so their club $s \equiv_c \texttt{bot}$. Since FWHNFs are only comparable to FWHNFs and `bot` (lemma 2.112), all the s_i with $i \geq i_0$ must have FWHNFs.

3. $D \equiv \texttt{c}\ D_1 \ldots D_{\alpha(\texttt{c})}$: As in 2. there must be an $i_0 : s_{i_0}$ has an SCWHNF $\texttt{c}\ t_1^{i_0} \ldots t_{\alpha(\texttt{c})}^{i_0}$. Then, as a consequence of lemma 2.112, for all $i \geq i_0 : s_i {\Downarrow_S} \texttt{c}\ t_1^i \ldots t_{\alpha(\texttt{c})}^i$. By lemma 2.95 the $t_j^{i_0} \leq_c t_j^{i_0+1} \leq_c \ldots$ form ascending chains and by theorem 2.125 the chains satisfy $\bigsqcup_i^c t_j^i$ in $\Lambda_p(D_j)$. By structural induction we conclude that for every j all but finitely many elements t_j^i are in $\Lambda_p(D_j)$. Summarizing our claim follows. □

Lemma 3.17. *There are* $s \in \Lambda_p(D)$ *with* $s \equiv_c \bigsqcup_i^c s_i$ *and* $\forall i : s_i \not\equiv_c s_{i+1}$.

Proof. We present an example. $s_i \overset{\text{def}}{=} \lambda x.(x\texttt{:})^i\ \texttt{bot}$. While $s_i \leq_c s_{i+1}$ we have $s_{i+1} \not\leq_c s_i$. Anticipating proposition 3.52, equation (3.11) $\bigsqcup_i^c \lambda x.((x\texttt{:})^i\ \texttt{bot}) \equiv_c \lambda x.(\texttt{repeat}\ x)$ and with continuity and $\mathbb{C}[\cdot] \overset{\text{def}}{=} \lambda x.[\cdot]$ we obtain $\bigsqcup_i^c s_i \equiv_c \bigsqcup_i^c \mathbb{C}[(x\texttt{:})^i\ \texttt{bot}] \equiv_c \mathbb{C}[\bigsqcup_i^c (x\texttt{:})^i\ \texttt{bot}] \equiv_c \mathbb{C}[\texttt{repeat}\ x] \equiv_c \lambda x.(\texttt{repeat}\ x) \in \Lambda_p(\texttt{Fun})$. □

$\Lambda_p(\cdot)$ covers only a small subset of the demands, the most remarkable syntactic features that were not yet covered are recursive use of demands and bindings using `where`-expressions. A demand definition, which makes use of the names of demands recursively is e.g. $\texttt{Inf} = \langle\texttt{Bot}, \texttt{Top} : \texttt{Inf}\rangle$. We need to define approximations to the representation in such a way that they use functions ρ mapping demand names to subsets of Λ. The function Δ improves the mapping of names to subsets of Λ so that eventually we can define the representation of a demand with the fixed point of Δ.

Definition 3.18. Let $\rho : N(\mathcal{C}) \to \mathcal{P}(\Lambda)$. We define $\Sigma(T) \overset{\text{def}}{=} \{\sigma | \sigma T \text{ is a primitive demand}\}$. Furthermore, we define $\eta_\rho : \Lambda_C \to \mathcal{P}(\Lambda)$ as the function for which:

$$\eta_\rho(\texttt{Bot}) \overset{\text{def}}{=} \Lambda_p(\texttt{Bot}) \tag{3.4}$$

$$\eta_\rho(\texttt{Fun}) \overset{\text{def}}{=} \Lambda_p(\texttt{Fun}) \tag{3.5}$$

$$\eta_\rho(\texttt{c}\ C_1 \dots C_{\alpha(\texttt{c})}) \overset{\text{def}}{=} \{t | t \Downarrow_S \texttt{c}\ \vec{t} \wedge \forall i : t_i \in \eta_\rho(C_i)\} \tag{3.6}$$

$$\eta_\rho(\langle C_1, \dots, C_n\rangle) \overset{\text{def}}{=} \bigcup_i \eta_\rho(C_i) \tag{3.7}$$

$$\eta_\rho(C_1 \cap \dots \cap C_n) \overset{\text{def}}{=} \bigcap_i \eta_\rho(C_i) \tag{3.8}$$

$$\eta_\rho(N) \overset{\text{def}}{=} \rho(N), \text{ if } N \in N(\mathcal{C}) \tag{3.9}$$

$$\eta_\rho(S\ \texttt{where}\ T = N) \overset{\text{def}}{=} \bigcup_{\sigma \in \Sigma(T) : \Lambda_p(\sigma T) \subseteq \rho(N)} \eta_\rho(\sigma S) \tag{3.10}$$

η_ρ is an *approximation* of the representation for every ρ.

The approximations are improved by Δ.

Definition 3.19.

$$\Delta(\rho) \overset{\text{def}}{=} \rho', \text{ where } \rho'(N) \overset{\text{def}}{=} \eta_\rho(D), \text{ if } N = D \in \mathcal{C}.$$

Lemma 3.20. Δ *maps elements of* $L \overset{\text{def}}{=} \mathcal{P}(\Lambda)^{N(\mathcal{C})}$ *to elements of* L.

For a demand D the set of terms represented by D, $\eta(D)$, can now be defined with the smallest fixpoint of Δ.

Definition 3.21. $\eta(C) \overset{\text{def}}{=} \eta_\beta(C)$, where $\beta \overset{\text{def}}{=} \mu x.\Delta(x)$. We call $\eta(C)$ the set of *expressions represented* by C.

If Δ is monotone, it follows from the Theorem of Knaster and Tarski that this fixed point does indeed exist. We show monotonicity next.

Lemma 3.22. *Let* $\rho_1 \leq \rho_2$, *then*

$$\Delta(\rho_1) \leq \Delta(\rho_2).$$

Proof. We show $\forall (M = D) \in \mathcal{C} : \eta_{\rho_1}(D) \subseteq \eta_{\rho_2}(D)$ using induction on pairs (w, D) for every demand D. In these, w is the number of `where`-expressions in D and for D we use the structural order.

$D \equiv N$: $\eta_{\rho_1}(N) = \rho_1(N)$ and the premise states $\rho_1(N) \subseteq \rho_2(N) = \eta_{\rho_2}(N)$.

$D \in \{\texttt{Bot}, \texttt{Fun}\}$: $\eta_{\rho_1}(D) = \Lambda_p(D)$ and this is true for arbitrary ρ_1, in particular $\Lambda_p(D) = \eta_{\rho_2}(D)$.

$D \equiv \texttt{c}\ D_1 \ldots D_{\alpha(\texttt{c})}$:

$$\begin{aligned}
&\eta_{\rho_1}(D)\\
=&(\text{definition 3.18})\\
&\{t | t \Downarrow_S \texttt{c}\ t_1 \ldots t_{\alpha(\texttt{c})} \wedge \forall i : t_i \in \eta_{\rho_1}(D_i)\}\\
\subseteq&(\text{induction hypothesis})\\
&\{t | t \Downarrow_S \texttt{c}\ t_1 \ldots t_{\alpha(\texttt{c})} \wedge \forall i : t_i \in \eta_{\rho_2}(D_i)\}\\
=&(\text{definition 3.18})\\
&\eta_{\rho_2}(D).
\end{aligned}$$

$D \equiv \langle D_1, \ldots, D_n \rangle$: Analogously to $D \equiv \texttt{c}\ D_1 \ldots D_n$.

$D \equiv D_1 \cap \cdots \cap D_n$: ditto

$D \equiv S\ \texttt{where}\ T = N$:

$$\begin{aligned}
&\eta_{\rho_1}(D)\\
=&(\text{definition 3.18})\\
&\bigcup_{\sigma : \Lambda_p(\sigma T) \subseteq \eta_{\rho_1}(N)} \eta_{\rho_1}(\sigma S)\\
\subseteq&(\text{induction hypothesis: } \sigma S \text{ has one } \texttt{where} \text{ less than}\\
&\quad D + \text{premise})\\
&\bigcup_{\sigma : \Lambda_p(\sigma T) \subseteq \eta_{\rho_2}(N)} \eta_{\rho_2}(\sigma S)\\
=&(\text{definition 3.18})\\
&\eta_{\rho_2}(D).
\end{aligned}$$

Hence Δ is monotone. □

For many proofs it will be helpful to have an iterative characterization of β. Such a characterization can be obtained by the CPO Fixpoint Theorem I [DP90]. This theorem requires a monotone function on a CPO, then if $\alpha \overset{\text{def}}{=} \bigsqcup_{i \geq 0} \Delta^i(\mathbf{0})$ is a fixed point, α is also the least fixed point.
Since we have already shown monotonicity of Δ in lemma 3.22 and since every complete lattice is a CPO, so $\mathcal{P}(\Lambda)^{N(\mathcal{C})}$ is a CPO, all that remains to be shown is that α is a fixed point of Δ.

Lemma 3.23. $\bigsqcup_{i \geq 0} \Delta^i(\mathbf{0}) = \Delta(\bigsqcup_{i \geq 0} \Delta^i(\mathbf{0}))$.

Proof. We show the two inequalities separately.

$\leq$: $\mathbf{0} \leq \Delta(\mathbf{0})$, so monotonicity suffices to show this inequality.

$\geq$: We show $\forall x : \Delta(\bigsqcup_{i \geq 0} \Delta^i(\mathbf{0}))(x) \subseteq (\bigsqcup_{i \geq 0} \Delta^i(\mathbf{0}))(x)$.

Let $y \in \Delta(\bigsqcup_{i \geq 0} \Delta^i(\mathbf{0}))(x)$ then there is a j for which $y \in \Delta(\bigsqcup_{0 \leq i \leq j} \Delta^i(\mathbf{0}))(x) = \Delta(\Delta^j(\mathbf{0}))(x) = \Delta^{j+1}(\mathbf{0})(x)$. The first equality is due to $i < j \implies \Delta^i(\mathbf{0}) \leq \Delta^j(\mathbf{0})$. Obviously, $\Delta^{j+1}(\mathbf{0})(x) \subseteq \bigcup_{i \geq 0}(\Delta^i(\mathbf{0})(x)) = (\bigsqcup_{i \geq 0} \Delta^i(\mathbf{0}))(x)$ holds and so $y \in (\bigsqcup_{i \geq 0} \Delta^i(\mathbf{0}))(x)$. □

Since all the requirements are met, we obtain the following lemma from the CPO Fixpoint Theorem I:

Lemma 3.24. $\mu\chi.\Delta(\chi) = \bigsqcup_{i \geq 0} \Delta^i(\mathbf{0})$.

For notational convenience we will call $\eta_{\Delta^i(\mathbf{0})}(D)$ the *ith iteration of the representation* of D.

As a consequence of lemma 3.24 we will be able to use induction to prove properties of elements in the representation of demands. The first such proof is that of the following lemma and we will hereafter call induction using this particular measure *representational induction.*

Lemma 3.25. *$w \in \eta(D) \implies \exists E : w \in \Lambda_p(E)$, i.e. expressions in the representation are primitive.*

Proof. The statement is proved by induction on triples, where the first component is the least i for which $w \in \eta_{\Delta^i(\mathbf{0})}(D)$, the second component is the number of `where`s in D, written $w(D)$, and the third component is the demand D.
Here demands are ordered syntactically and the the triples are ordered lexicographically with the demand in the least significant position.

$D \in \{\texttt{Bot}, \texttt{Fun}\}$: by definition $\forall i : \eta_{\Delta^i(\mathbf{0})}(D) = \Lambda_p(D)$.

$D \equiv \texttt{c}\ D_1 \dots D_{\alpha(\texttt{c})}$: $w \in \eta_{\Delta^i(\mathbf{0})}(D) \iff w \Downarrow_S \texttt{c}\ w_1 \dots w_{\alpha(\texttt{c})} \wedge \forall j : w_j \in \eta_{\Delta^i(\mathbf{0})}(D_j)$. For the w_j the first component remains unchanged, the second will not grow and the third will surely be smaller, thus $(i, w(D_j), D_j) < (i, w(D), D)$. With the induction hypothesis we conclude that $E_1, \dots, E_{\alpha(\texttt{c})}$ exist, with $w_j \in \Lambda_p(E_j)$ so $w \in \Lambda_p(\texttt{c}\ E_1 \dots E_{\alpha(\texttt{c})})$.

$D \equiv \langle D_1, \dots, D_n \rangle$: For at least one of the $D_j : w \in \eta_{\Delta^i(\mathbf{0})}(D_j)$. Again the third component definitely decreases and the second does not increase. With the induction hypothesis we conclude that E_j exists with $w \in \Lambda_p(E_j)$.

$D \equiv D_1 \cap \dots \cap D_n$: For every $D_j : w \in \eta_{\Delta^i(\mathbf{0})}(D_j)$, so the statement follows from the induction hypothesis.

$D \equiv N$: We have $w \in \Delta^i(\mathbf{0})(N) = \eta_{\Delta^{i-1}(\mathbf{0})}(S)$, if $N = S \in \mathcal{C}$. The first component becomes smaller and by induction hypothesis the statement holds.

$D \equiv S\ \texttt{where}\ T = N$: If $w \in \eta_{\Delta^i(\mathbf{0})}(D)$, then there is a σ for which $w \in \eta_{\Delta^i(\mathbf{0})}(\sigma S)$ and σ does not substitute `where`-expressions. $w(\sigma S) < w(D)$ and the first component remains unchanged. The induction hypothesis then implies our claim. □

From the above lemma we know that every expression represented is a primitive expression since some primitive demand for it exists. Now we establish the fact that one expression from some primitive demand's expressions suffices for all of its primitive expressions to be included in a single iteration of the representation.

Lemma 3.26. *Let D be a demand and let E be a primitive demand, then*

$$\eta(D) \cap \Lambda_p(E) \neq \emptyset \implies \exists i : \Lambda_p(E) \subseteq \eta_{\Delta^i(\mathbf{0})}(D).$$

Proof. The proof uses representational induction as does the proof of lemma 3.25. In the cases that $D \equiv \texttt{Bot}, D \equiv \texttt{Fun}$ or $D \equiv \texttt{c}\ D_1 \dots D_{\alpha(\texttt{c})}$ we infer that $E \equiv \texttt{Bot}, E \equiv \texttt{Fun}$ or $E \equiv \texttt{c}\ E_1 \dots E_{\alpha(\texttt{c})}$, respectively. □

Corollary 3.27. *Let D be a demand and let E be a primitive demand, then*

$$\Lambda_p(E) \not\subseteq \eta(D) \implies \eta(D) \cap \Lambda_p(E) = \emptyset.$$

Proof.

$$\begin{aligned}
&\eta(D) \cap \Lambda_p(E) \neq \emptyset \\
\implies &\text{(lemma 3.26)} \\
&\exists i : \Lambda_p(E) \subseteq \eta_{\Delta^i(\mathbf{0})}(D) \\
\implies & \\
&\Lambda_p(E) \subseteq \bigcup_i \eta_{\Delta^i(\mathbf{0})}(D) = \eta(D).
\end{aligned}$$

□

Intuitively, either one or all of the primitive terms in a primitive demand are present in the representation of a given demand (lemma 3.26 and corollary 3.27).
The set of expressions associated with a primitive demand is closed with respect to $\equiv_c$.

Lemma 3.28. *Let D be a demand and let $s, t \in \Lambda$ with $s \equiv_c t$, then*

$$s \in \Lambda_p(D) \iff t \in \Lambda_p(D).$$

Proof. We use induction on the structure of D and show one implication only, the other one can be obtained by renaming.

$D \equiv \texttt{Bot}$: Lemma 2.108 implies $s\Uparrow \wedge t\Uparrow$, so $t \in \Lambda_p(D)$.

$D \equiv \texttt{Fun}$: Assume $s\Downarrow_F$ but $t \not\Downarrow_F$. Lemma 2.107 implies $\forall r \in \Lambda : t \not\equiv_c \lambda x.r$, but $s \equiv_c \lambda x.(sx)$. So we have $s \not\equiv_c t$ in contradiction to our assumption.

$D \equiv \texttt{c}\ D_1 \ldots D_{\alpha(\texttt{c})}$: From the induction hypothesis we obtain $s_i \in \Lambda_p(D_i) \wedge s_i \equiv_c t \implies t_i \in \Lambda_p(D_i)$. With lemma 2.112

we conclude $s\Downarrow_S \texttt{c}\ s_1 \ldots s_{\alpha(\texttt{c})} \wedge s \equiv_c t \implies t\Downarrow_S \texttt{c}\ t_1 \ldots t_{\alpha(\texttt{c})} \wedge \forall i : t_i \equiv_c s_i$. This implies $s \in \Lambda_p(D) \wedge s \equiv_c t \implies t \in \Lambda_p(D)$. □

The same property can be shown for η_ρ if we require all the sets of Λ-expressions that ρ associates to names to be $\equiv_c$-closed.

Lemma 3.29. *Let $\rho : N(\mathcal{C}) \to \mathcal{P}(\Lambda)$ where every element in the co-domain of ρ is closed with respect to $\equiv_c$, let $s, t \in \Lambda$ with $s \equiv_c t$ and let D be a demand.*

$$s \in \eta_\rho(D) \iff t \in \eta_\rho(D).$$

Proof. Use induction on lexicographically ordered pairs having the number of `where`-expressions in their first component and the demand in their second. For the second component the structural order is assumed. The base case is provided by lemma 3.28. □

Corollary 3.30. *Let D be a demand and $s, t \in \Lambda$, $s \equiv_c t$, then $\exists i : s \in \eta_{\Delta^i(\mathbf{0})}(D) \iff t \in \eta_{\Delta^i(\mathbf{0})}(D)$.*

Proof. Let $s \in \eta(D) \wedge s \equiv_c t$, then by lemma 3.25 there is a demand E for which $s \in \Lambda_p(E)$, by lemma 3.28 $t \in \Lambda_p(E)$ and so $\exists i : t \in \eta_{\Delta^i(\mathbf{0})}(D) \wedge s \in \eta_{\Delta^i(\mathbf{0})}(D)$ since by lemma 3.26 $\Lambda_p(E) \subseteq \eta(D)$. The proof also works for the other inclusion. □

Corollary 3.31. *Let D be a demand and $s, t \in \Lambda$ with $s \equiv_c t$, then*

$$s \in \eta(D) \iff t \in \eta(D).$$

Lemma 3.26 implies that either all or none of the primitive terms of a primitive demand are included in the representation of a demand. Therefore one possibility to state which Λ-expressions are included in the representation of a demand D is to provide primitive demands with a representation included in that of D. We will make frequent use of this possibility and hence define the following notation.

Definition 3.32 ($\sqsubseteq_p^i$, primitively included). Let D be a demand, and let D_p be a primitive demand. If $\Lambda_p(D_p) \subseteq \eta_{\Delta^i(\mathbf{0})}(D)$, then we write $D_p \sqsubseteq_p^i D$ which is pronounced D_p *is primitively included in iteration* i *of* D. Accordingly we write $D_p \sqsubseteq_p D$ for $\Lambda_p(D_p) \subseteq \eta(D)$ and say D_p *is primitively included in* D.

Definition 3.33. $C \leq_\eta D \iff \eta(C) \subseteq \eta(D)$. We write $C = D$, iff $C \leq_\eta D$ and $D \leq_\eta C$.

As for Λ-expressions we will use contexts for demand expressions. *Demand-contexts* will be constructed from demands similarly to the way Λ-contexts are constructed from Λ-expressions. The same holds for the other definitions, e.g. depth. Whenever it seems appropriate for ease of comprehension, we will be explicit and speak about Λ-contexts and demand-contexts or D-contexts, in other places we will, if the nature of the context is evident from the textual context, in favor of readability speak only of contexts.

Lemma 3.34. *The $\leq_\eta$-relation on demands is a precongruence.*

Proof. The proof is in three parts:

$\leq_\eta$ is reflexive: obvious.

$\leq_\eta$ is transitive: obvious.

$C \leq_\eta D \implies \mathbb{C}[C] \leq_\eta \mathbb{C}[D]$ for arbitrary demand contexts $\mathbb{C}[\cdot]$: For this part we use induction on the depth d of the context $\mathbb{C}$.

$d = 0:$ $\mathbb{C}[\cdot] \equiv [\cdot]$ satisfies the hypothesis.

$d > 0:$ If the hypothesis holds for all depths $d < n$, then because of the covariant definition of η_ρ it also holds for depth n. □

One problem is immediately apparent: if the demand expressions are to be extended to provide a function space constructor $\overset{\forall}{\longrightarrow}$, due to its contra-variance, monotonicity of Δ is lost. We can regain monotonicity by requiring the left operand of $\overset{\forall}{\longrightarrow}$ not to contain any names. If one wishes to allow names on the left hand side, the existence of a fixed point for a non-monotone Δ will have to be proven. This is outside the scope of this work.

Lemma 3.35. *Let T and U be patterns without common variables, then the following holds:*

$$\eta((S \texttt{ where } T = N) \texttt{ where } U = M) = \eta((S \texttt{ where } U = M) \texttt{ where } T = N).$$

Proof.

$$\begin{aligned}
&\eta((S \texttt{ where } T = N) \texttt{ where } U = M)\\
&= \bigcup_{\sigma\in\Sigma(U):\Lambda_p(\sigma U)\subseteq\eta(M)} \eta(\sigma(S \texttt{ where } T = N))\\
&= \bigcup_{\sigma\in\Sigma(U):\Lambda_p(\sigma U)\subseteq\eta(M)} \left(\bigcup_{\rho\in\Sigma(\sigma T):\Lambda_p(\rho\sigma T)\subseteq\eta(N)} \eta(\rho\sigma S)\right)\\
&= \bigcup_{\substack{\pi:\exists\rho\in\Sigma(\sigma T),\sigma\in\Sigma(U)\wedge\rho\sigma=\pi\\ \wedge\Lambda_p(\pi U)\subseteq\eta(M)\wedge\Lambda_p(\pi T)\subseteq\eta(N)}} \eta(\pi S)
\end{aligned}$$

$\subseteq$: The last step can be seen as follows: ρ as well as σ can bind variables from U or T, respectively. But then we may rewrite

$\rho\sigma = \rho'\sigma'$, where σ' does not bind variables from T and ρ' does not bind variables from U and $\rho'_{|T} = \sigma_{|T}$ and $\sigma'_{|U} = \rho_{|U}$. Since ρ' and σ' appear in the unions above and $\rho'\sigma'U = \sigma U = \sigma' U$ and $\rho'\sigma'T = \rho\sigma T = \rho' T$, we may perform the last step.

$\supseteq$: Given π on the other hand and a disjoint set of variables in $T \cup U$, we may write $\pi = \rho\sigma$, such that σ does not substitute variables from T and ρ none from U. Then $\rho\sigma T \equiv \rho T \equiv \pi T$ and $\sigma T \equiv T$ and the statement ensues. ☐

Notation 3.36. As a consequence we may subsequently write and regard as equivalent:

$$S \texttt{ where } T = N; U = M \text{ and } S \texttt{ where } U = M; T = N$$

if there are no common variables in T and U.

Lemma 3.37. *Let $\mathtt{c} \neq \mathtt{c}'$ be constructors. Then for any demands $D'_i, D_i : \eta(\mathtt{c}\ D_1 \ldots D_{\alpha(\mathtt{c})}) \cap \eta(\mathtt{c}'\ D'_1 \ldots D'_{\alpha(\mathtt{c}')}) = \emptyset$.*

Proof. Assume this does not hold. Then there is a $t \in \Lambda$ with $t \in \eta(\mathtt{c}\ D_1 \ldots D_{\alpha(\mathtt{c})})$ and $t \in \eta(\mathtt{c}'\ D'_1 \ldots D'_{\alpha(\mathtt{c}')})$. Let i and i' be the least numbers of iterations satisfying $t \in \eta_{\Delta^i(\mathbf{0})}(\mathtt{c}\ D_1 \ldots D_{\alpha(\mathtt{c})})$ and $t \in \eta_{\Delta^{i'}(\mathbf{0})}(\mathtt{c}'\ D'_1 \ldots D'_{\alpha(\mathtt{c}')})$. It follows that $t{\Downarrow}\mathtt{c}\ t_1 \ldots t_{\alpha(\mathtt{c})}$ and $t{\Downarrow}\mathtt{c}'\ t'_1 \ldots t'_{\alpha(\mathtt{c}')}$. It is easily seen from the definition of $\rightarrow$ that this is impossible. ☐

Some simple properties of η.

Lemma 3.38. *Let C_i be demands. η satisfies:*

1. $\eta(\mathtt{Bot}) = \{t | t{\Uparrow}\}$
2. $\eta(\mathtt{c}\ C_1 \ldots C_{\alpha(\mathtt{c})}) = \{t | t{\Downarrow}\mathtt{c}\ t_1 \ldots t_{\alpha(\mathtt{c})} \wedge \forall i : t_i \in \eta(C_i)\}$

3. $\eta(\langle C_1, C_2\rangle) = \eta(C_1) \cup \eta(C_2)$
4. $\eta(C_1 \cap C_2) = \eta(C_1) \cap \eta(C_2)$
5. $\forall \sigma \in \Sigma(T) : \sigma T \sqsubseteq_p N \implies \eta(\sigma S) \subseteq \eta(S \texttt{ where } T = N)$.

Proof.

1. $\eta(\mathtt{Bot}) = \eta_\beta(\mathtt{Bot})$, but $\forall \rho : \eta_\rho(\mathtt{Bot}) = \Lambda_p(\mathtt{Bot}) = \{t | t{\Uparrow}\}$.
2. $\subseteq$: If $w \in \eta(\mathtt{c}\ C_1 \ldots C_{\alpha(\mathtt{c})})$ then $\exists i : w \in \eta_{\Delta^i(\mathbf{0})}(\mathtt{c}\ C_1 \ldots C_{\alpha(\mathtt{c})}) = \{t | t{\Downarrow}\mathtt{c}\ t_1 \ldots t_{\alpha(\mathtt{c})} \wedge \forall j : t_j \in \eta_{\Delta^i(\mathbf{0})}(C_j)\}$. With lemma 3.23 it follows that $\eta_{\Delta^i(\mathbf{0})}(C_j) \subseteq \eta(C_j)$ and thus $w \in \{t | t{\Downarrow}\mathtt{c}\ t_1 \ldots t_{\alpha(\mathtt{c})} \wedge \forall j : t_j \in \eta(C_j)\}$.

   $\supseteq$: If $w \in \{t | t{\Downarrow}\mathtt{c}\ t_1 \ldots t_{\alpha(\mathtt{c})} \wedge \forall j : t_j \in \eta(C_j)\}$ then $w{\Downarrow}\mathtt{c}\ t_1 \ldots t_{\alpha(\mathtt{c})} \wedge \forall j \exists i_j : t_j \in \eta_{\Delta^{i_j}(\mathbf{0})}(C_j)$ and with lemma 3.23 we obtain for $i = \max_j i_j : \forall j : t_j \in \eta_{\Delta^i(\mathbf{0})}(C_j)$ implying $w \in \eta_{\Delta^i(\mathbf{0})}(\mathtt{c}\ C_1 \ldots C_{\alpha(\mathtt{c})}) \subseteq \eta(\mathtt{c}\ C_1 \ldots C_{\alpha(\mathtt{c})})$.
3. $\subseteq$: If $w \in \eta(\langle C_1, C_2\rangle)$ then $\exists i : w \in \eta_{\Delta^i(\mathbf{0})}(\langle C_1, C_2\rangle) = \eta_{\Delta^i(\mathbf{0})}(C_1) \cup \eta_{\Delta^i(\mathbf{0})}(C_2) \subseteq \eta(C_1) \cup \eta(C_2)$.

   $\supseteq$: If $w \in \eta(C_1) \cup \eta(C_2)$ then $\exists i : w \in \eta_{\Delta^i(\mathbf{0})}(C_1) \vee \exists i : w \in \eta_{\Delta^i(\mathbf{0})}(C_2) \implies \exists i : w \in \eta_{\Delta^i(\mathbf{0})}(C_1) \cup \eta_{\Delta^i(\mathbf{0})}(C_2) = \eta_{\Delta^i(\mathbf{0})}(\langle C_1, C_2\rangle) \subseteq \eta(\langle C_1, C_2\rangle)$.
4. analogous to 3
5. $\sigma T \sqsubseteq_p N \iff \Lambda_p(\sigma T) \subseteq \eta(N)$. By lemma 3.26 $\exists i : \Lambda_p(\sigma T) \subseteq \eta_{\Delta^i(\mathbf{0})}(N) = \Delta^i(\mathbf{0})(N)$. If $w \in \eta_{\Delta^j(\mathbf{0})}(\sigma S)$ we choose $k = \mathit{max}\{i, j\}$ and obtain $w \in \bigcup_{\sigma \in \Sigma(T) : \sigma T \sqsubseteq_p^k N} \eta_{\Delta^k(\mathbf{0})}(\sigma S) \subseteq \eta(S \texttt{ where } T = N)$. ☐

Notation 3.39. $\eta(\langle\rangle)$, and as we will later see $\gamma(\langle\rangle)$, is empty. So we will also write $\emptyset$ for $\langle\rangle$, since it captures the intuition.

3.3 Concretization

The next definition introduces the notion of *concretization*, which is central to this work. It is the club-closure of $\eta(\cdot)$.

Definition 3.40 (Concretization). The *concretization* of a demand expression, C, is

$$\gamma(C) \overset{\text{def}}{=} \bigsqcup^c \eta(C).$$

Lemma 3.41. *Let D be a demand and let $s, t \in \Lambda$ with $s \equiv_c t$, then $s \in \gamma(D) \iff t \in \gamma(D)$.*

Primitive expressions in the concretization of a demand are already present in its representation. The concretization thus does not add any primitive expressions but only those which cannot be in the representation of any demand.

Theorem 3.42. *Let t be a primitive expression, then $t \in \gamma(D) \implies t \in \eta(D)$.*

Proof. In lemma 3.72 we have shown that the set of primitive positions, $\Pi(t)$, is finite and that at none of the primitive positions there is an expression reducing to $\mathbb{R}[x]$.
Now $t \in \gamma(D)$ and thus there is an ascending chain $t_i \leq_c t_{i+1}$ of expressions in $\eta(D)$ with $t \equiv_c \bigsqcup^c_i t_i$. We use continuity of contexts to obtain the equivalence $g_{t,\pi}\ t \equiv_c g_{t,\pi}\ (\bigsqcup^c_i t_i) \equiv_c \bigsqcup^c_i (g_{t,\pi}\ t_i)$ for all $\pi \in \Pi(t)$. Since $g_{t,\pi}\ t \equiv_c$ `1` not all $g_{t,\pi}\ t_i$ can be equivalent to `bot` and for sufficiently large $i : \pi \in \Pi(t_i)$, so $\exists i_0 : \forall i > i_0 : \Pi(t_i) = \Pi(t)$.

We consider the maximal primitive positions π of t and distinguish these cases:

$t_{|\pi}\Uparrow$: It must be that $t_{i|\pi}\Uparrow$ and therefore $t \leq_\pi t_i$.

$t_{|\pi}\Downarrow_F$: If for all the t_i with $i > i_0$ we had $t_{i|\pi}\Uparrow$ then t would not be equivalent to $\bigsqcup^c_i t_i$, since we could replace the manifest position π in suitable contracti of t with `bot` and obtain a $t' <_c t$ for which $t' \equiv_c \bigsqcup^c_i t_i$. It follows that for sufficiently large $i : t_{i|\pi}\Downarrow_F$ and thus $t \leq_\pi t_i$.

$t_\pi\Downarrow_S$`c` : The argument for $t \leq_\pi t_i$ for sufficiently large i is similar to that for $t_{|\pi}\Downarrow_F$.

It follows that for sufficiently large $i : \forall\pi \in \Pi(t) : t \leq_\pi t_i$ and therefore $t \leq_c t_i$ and $t \in \eta(D)$. □

Properties of γ

Some simple properties of this definition follow.

Lemma 3.43.

1. $\gamma(\texttt{Bot}) = \{t | t\Uparrow\}$
2. $\gamma(\texttt{c}\ C_1 \ldots C_{\alpha(\texttt{c})}) = \{t | t\Downarrow_S \texttt{c}\ t_1 \ldots t_{\alpha(\texttt{c})} | t_i \in \gamma(C_i)\}$
3. $\gamma(\langle C_1, C_2\rangle) = \gamma(C_1) \cup \gamma(C_2)$
4. $\gamma(C_1 \cap C_2) \subseteq \gamma(C_1) \cap \gamma(C_2)$
5. *In general* $\gamma(C_1 \cap C_2) \not\supseteq \gamma(C_1) \cap \gamma(C_2)$
6. $\forall\sigma \in \Sigma(T) : \sigma T \sqsubseteq_p N \implies \gamma(\sigma S) \subseteq \gamma(S\ \texttt{where}\ T = N)$.

Proof.

1. By corollary 2.109 all expressions in $\eta(\texttt{Bot})$ are $\equiv_c$-equivalent and so is their club.

2. $\subseteq$: In ascending chains of expressions from $\eta(\texttt{c}\ C_1 \dots C_{\alpha(\texttt{c})})$ every expression is $\equiv_c$-equivalent to an expression starting with constructor `c`. This is a consequence of lemma 2.95. Theorem 2.125 implies that we can perform the club operation component-wise and this yields the statement.

   $\supseteq$: This inclusion is proved just like the other.

3. $\supseteq$: Without loss of generality $t \in \gamma(C_1) = \bigsqcup^c \eta(C_1)$. From lemma 3.38 we get $\eta(C_1) \subseteq \eta(C_1) \cup \eta(C_2) = \eta(\langle C_1, C_2\rangle)$. Applying the club results in $\bigsqcup^c \eta(C_1) \subseteq \bigsqcup^c \eta(\langle C_1, C_2\rangle) = \gamma(\langle C_1, C_2\rangle)$.

   $\subseteq$: $t \in \gamma(\langle C_1, C_2\rangle) \iff t \equiv_c \bigsqcup^c_i t_i \wedge \forall i : t_i \in \eta(\langle C_1, C_2\rangle) = \eta(C_1) \cup \eta(C_2)$. Infinitely many of the chains elements come from one of the two representations, without loss of generality from $\eta(C_1)$. There is $I = \{i_1, i_2, \dots\}$ with $|I| = \infty \wedge \forall i \in I : t_i \in \eta(C_1)$. By lemma 2.121 we obtain $\bigsqcup^c_{i\in I} t_i \equiv_c t$ and thus $t \in \gamma(C_1) \subseteq \gamma(C_1) \cup \gamma(C_2)$.

4. $t \in \gamma(C_1 \cap C_2) \iff \bigsqcup^c_i t_i \equiv_c t \wedge \forall i : t_i \in \eta(C_1 \cap C_2) = \eta(C_1) \cap \eta(C_2)$. Then $t \in \bigsqcup^c \eta(C_1)$ and $t \in \bigsqcup^c \eta(C_2)$, so $t \in \gamma(C_1) \cap \gamma(C_2)$.

5. Define $\texttt{Inf1Even} \overset{\text{def}}{=} \langle \texttt{Bot}, \texttt{1 : 1 : Inf1Even}\rangle$ and $\texttt{Inf1Odd} \overset{\text{def}}{=} \langle \texttt{1 : Bot}, \texttt{1 : 1 : Inf1Odd}\rangle$. A straightforward inductive argument shows all lists in $\eta(\texttt{Inf1Even})$ to have an even number of `1`s and all lists in $\eta(\texttt{Inf1Odd})$ to have an odd number of `1`s, hence $\eta(\texttt{Inf1Even} \cap \texttt{Inf1Odd}) = \emptyset$ which implies $\gamma(\texttt{Inf1Even} \cap \texttt{Inf1Odd}) = \emptyset$. But $\texttt{repeat 1} \in \gamma(\texttt{Inf1Even})$ and $\texttt{repeat 1} \in \gamma(\texttt{Inf1Odd})$. A fact that can be seen e.g. from lemma 2.121 since $\eta(\texttt{Inf1Even})$ as well as $\eta(\texttt{Inf1Odd})$ contain infinitely many of the elements of the chain $(\texttt{1:})^i\ \texttt{bot}$ which we have shown to have club `repeat 1` in proposition 3.52.

6. Due to theorem 3.42 we know $\Lambda_p(\sigma T) \subseteq \gamma(N) \iff \Lambda_p(\sigma T) \subseteq \eta(N)$. From lemma 3.38 we obtain $\eta(S\ \texttt{where}\ T = N) \supseteq \eta(\sigma S)$ and by monotonicity of clubs $\bigsqcup^c \eta(S\ \texttt{where}\ T = N) \supseteq \bigsqcup^c \eta(\sigma S)$. □

Lemma 3.44. *The demand definitions for M and N in the following table have the same concretization.*

M	N
$\langle \texttt{c}\ A\ X, \texttt{c}\ A\ Y\rangle$	$\texttt{c}\ A\ \langle X, Y\rangle$
$\langle A \cap X, A \cap Y\rangle$	$A \cap \langle X, Y\rangle$
$\langle \texttt{c}\ A\ X\ \texttt{where}\ T = L,$ $\texttt{c}\ A\ Y\ \texttt{where}\ T = L\rangle$	$\texttt{c}\ A\ \langle X, Y\rangle\ \texttt{where}\ T = L$
$\langle\langle C_1, \dots, C_n\rangle\rangle$	$\langle C_1, \dots, C_n\rangle$

Proof. Because of theorem 2.124 it suffices to show $\eta(N) = \eta(M)$. Recall that $\eta_\beta(N) = \eta_\beta(M) \iff \beta(N) = \beta(M)$ and $\beta = \bigsqcup_i \Delta^i(\mathbf{0})$.

We show $\Delta^i(\mathbf{0})(N) = \Delta^i(\mathbf{0})(M)$ for $M \overset{\text{def}}{=} \langle \texttt{c}\ A\ X, \texttt{c}\ A\ Y\rangle$ and $N \overset{\text{def}}{=} \texttt{c}\ A\ \langle X, Y\rangle$. The other statements can be proven analogously.

$i = 0 : \ \Delta^0(\mathbf{0})(N) = \{\} = \Delta^0(\mathbf{0})(M)$

$i > 0$:

$$\begin{aligned}
&\Delta^i(\mathbf{0})(N)\\
&\quad = \eta_{\Delta^{i-1}(\mathbf{0})}(\mathtt{c}\ A\ \langle X, Y\rangle)\\
&\quad = \{t | t \Downarrow_S \mathtt{c}\ t_1\ t_2 \wedge t_1 \in \eta_{\Delta^{i-1}(\mathbf{0})}(A) \wedge \eta_{\Delta^{i-1}(\mathbf{0})}(\langle X, Y\rangle)\}\\
&\quad = \{t | t \Downarrow_S \mathtt{c}\ t_1\ t_2 \wedge t_1 \in \eta_{\Delta^{i-1}(\mathbf{0})}(A) \wedge \eta_{\Delta^{i-1}(\mathbf{0})}(X) \cup \eta_{\Delta^{i-1}(\mathbf{0})}(Y)\}\\
&\quad = \{t | t \Downarrow_S \mathtt{c}\ t_1\ t_2 \wedge t_1 \in \eta_{\Delta^{i-1}(\mathbf{0})}(A) \wedge \eta_{\Delta^{i-1}(\mathbf{0})}(X)\}\\
&\qquad \cup \{t | t \Downarrow_S \mathtt{c}\ t_1\ t_2 \wedge t_1 \in \eta_{\Delta^{i-1}(\mathbf{0})}(A) \wedge \eta_{\Delta^{i-1}(\mathbf{0})}(Y)\}\\
&\quad = \eta_{\Delta^{i-1}(\mathbf{0})}(\mathtt{c}\ A\ X) \cup \eta_{\Delta^{i-1}(\mathbf{0})}(\mathtt{c}\ A\ Y)\\
&\quad = \eta_{\Delta^{i-1}(\mathbf{0})}(\langle \mathtt{c}\ A\ X, \mathtt{c}\ A\ Y\rangle)\\
&\quad = \Delta^i(\mathbf{0})(M) \quad \square
\end{aligned}$$

From now on we will often assume that unions appear only at the top level of a demand expression, if that assumption simplifies the presentation.
We have already mentioned the intuition for demands, namely that they represent sets of values to which a Λ-expression evaluates.
It is natural to ask whether the concretization of a demand is contained in that of another. This concept is formalized in the definition of the $\leq_\gamma$ relation for demands.

Definition 3.45. Let C, D be demands.

$$C \leq_\gamma D \overset{\text{def}}{\iff} \gamma(C) \subseteq \gamma(D).$$

Lemma 3.46. *For demands C, D*

$$C \leq_\eta D \iff C \leq_\gamma D.$$

Proof.

$\Longrightarrow$: This follows from monotonicity of clubs.

$\Longleftarrow$: If $\gamma(C) \subseteq \gamma(D)$ and $t \in \eta(C)$ then $t \in \gamma(D)$ and by theorem 3.42 we conclude $t \in \eta(D)$. $\square$

We define a top element with respect to $\leq_\gamma$ for the demands and call it `Top`.

Definition 3.47 (`Top`). Let $\{A_1, \ldots, A_n\} = \mathcal{A}$.

$$\mathtt{Top} = \langle \mathtt{Bot}, \mathtt{Fun}, \mathtt{c}_{A_1,1}\ \underbrace{\mathtt{Top}\ldots\mathtt{Top}}_{\alpha(\mathtt{c}_{A_1,1})}, \ldots, \mathtt{c}_{A_n,|A_n|}\ \underbrace{\mathtt{Top}\ldots\mathtt{Top}}_{\alpha(\mathtt{c}_{A_n,|A_n|})}\rangle.$$

Proposition 3.48. `Top` *is a greatest element in Λ_C with respect to $\leq_\gamma$.*

Proof. $D \leq_\gamma \mathtt{Top} \iff D \leq_\eta \mathtt{Top}$ according to lemma 3.46. $\forall D \in \Lambda_C : w \in \eta(D) \implies w \in \eta(\mathtt{Top})$ can then be proved by representational induction. $\square$

We say a demand is `bot`-closed, if in every expression in its concretization an arbitrary subexpression can be replaced by `bot` resulting in an expression which also is in the concretization. Intuitively, this means that the concretization "contains `bot` at any depth".
Obviously, this property is undecidable, since e.g. $\mathtt{Bot} \leq_\gamma D$ could be reduced to it.
Later, we formulate rules for the calculi using such undecidable properties of demands, particularly in (loopdecomp), (loopred) and (noloop). Implementations will need to provide decidable approximations, which, however, will not be detailed here.

Definition 3.49. Let C be a demand. C is called `bot`-*closed*, iff:

$$\forall \mathbb{C}[\cdot], s : \mathbb{C}[s] \in \gamma(C) \implies \mathbb{C}[\mathtt{bot}] \in \gamma(C).$$

The demand $\langle\rangle$ is trivially `bot`-closed, since its concretization is empty.
It may look as if concretizations of `bot`-closed demands are down-sets, i.e. that for these concretizations, $\gamma(D)$ we have $\forall s \in \Lambda : t \in \gamma(D) \wedge s \leq_c t \implies s \in \gamma(D)$. This is not the case: while concretizations being down-sets are `bot`-closed, the converse is generally not true.

Lemma 3.50. *Let D be a demand, then*

$$\gamma(D) \textit{ is a down-set} \implies D \textit{ is } \texttt{bot}\textit{-closed.}$$

Proof. `bot` $\leq_c$ s for all s and since $\leq_c$ is a precongruence $\mathbb{C}[\texttt{bot}] \leq_c \mathbb{C}[s]$ and thus $\gamma(D)$ must contain $\mathbb{C}[\texttt{bot}]$ for every $\mathbb{C}[s]$ so D is `bot`-closed. □

The following example shows that, in general, the converse implication does not hold.

Example 3.51. $D \overset{\text{def}}{=} \langle \texttt{Bot}, \texttt{1}\rangle$ is `bot`-closed but $\gamma(D)$ is not down-closed.
$\lambda x.\texttt{bot} \leq_c \texttt{1}$ according to lemma 2.112 yet $\lambda x.\texttt{bot} \notin \gamma(D)$.
D is indeed `bot`-closed. Let $\mathbb{C}[s]$ be an arbitrary expression in $\gamma(D)$.

$\mathbb{C}[s]\Uparrow$: $\mathbb{C}[\texttt{bot}]\Uparrow$ since $\leq_c$ is a precongruence and `bot` $\leq_c$ s.

$\mathbb{C}[s]\Downarrow_S \texttt{1}$: We iterate the following distinction along the normal-order reduction sequence to `1`.

If $\mathbb{C}[s]$ and $\mathbb{C}[\texttt{bot}]$ already differ at argument depth 0, $\mathbb{C}[\cdot]$ must be a reduction context and thus $\mathbb{C}[\texttt{bot}]\Uparrow$, so $\mathbb{C}[\texttt{bot}] \in \gamma(D)$.

If $\mathbb{C}[s]$ and $\mathbb{C}[\texttt{bot}]$ are equal above argument depth $n > 0$, then we can reduce the same position in $\mathbb{C}[\texttt{bot}]$ that was reduced in the normal-order reduction of $\mathbb{C}[s]$ and this position will be the normal-order reduction redex.

It follows that: either $\mathbb{C}[\texttt{bot}]\Downarrow_S \texttt{1}$ or $\mathbb{C}[\texttt{bot}]\Uparrow$, so $\mathbb{C}[\texttt{bot}] \in \gamma(D)$.

The definition of γ is necessary in the sense that there are terms t in Λ for which there is a demand D such that $t \in \gamma(D)$, but for which there is no D such that $t \in \eta(D)$.

Proposition 3.52. `repeat 1` *is not primitive.*

Proof. `repeat 1` $\equiv \boldsymbol{\theta}$`(1 :)`. Consequently,

$$\texttt{repeat 1} \equiv_c \bigsqcup_i^c \texttt{(1 :)}^i \texttt{ bot} \tag{3.11}$$

by theorem 2.133.
$\forall i :$ `repeat 1` $\not\leq_c$ `(1 :)`^{i} `bot` and so `repeat 1` $\notin \eta(\texttt{Inf1})$ for the demand definition $\texttt{Inf1} = \langle \texttt{Bot}, 1 : \texttt{Inf1}\rangle$ since for all $s \in \eta(\texttt{Inf1})$ there is an $i \geq 0$ with $s \leq_c$ `(1 :)`^{i} `bot`. The latter can easily be proved by induction of the least number j of iterations of Δ necessary for $s \in \eta_{\Delta^i(\mathbf{0})}(\texttt{Inf1})$. But due to (3.11) `repeat 1` $\in \gamma(\texttt{Inf1})$. If `repeat 1` were primitive it would have to be in $\eta(\texttt{Inf1})$ according to theorem 3.42. □

3.4 Expressibility of Λ_C

We will show that the complexity of demands equals that of Turing-machines. Two separate directions show demands to be at least as complex as Turing-machines and Turing-machines to be at least as complex as demands. The former follows from reduction of Turing-machines to demands where the tape alphabet

is encoded in constructors such that one demand is primitively included in another iff the corresponding word is accepted by the Turing-machine. The presentation of an enumeration method for the demands primitively included in a given demand proves the latter direction. Primitive inclusion is appropriate, because either all or none of the primitive terms of a primitive demand are represented by a given demand (lemma 3.26 and corollary 3.27).
Demands and Turing-machines are equipotent. Consequently, even simple questions are undecidable for demands, e.g. emptiness or inclusion.
Note that demands with linear patterns, i.e. for which every variable in the pattern occurs exactly once, suffice for the reduction of Turing-machines to demands. So there is no change in the expressive power of demands if we restrict patterns to be linear.
The definition of Turing-machine is as in [HU79].

Definition 3.53. A *Turing-machine* M is defined as

$$M \overset{\text{def}}{=} (Q, \Sigma, \Gamma, \delta, q_0, B, F)$$

where

- Q is a finite set of states
- Γ is a finite set of tape symbols
- $B \in \Gamma$ is the blank symbol
- $\Sigma \subseteq \Gamma$ is the set of input symbols not containing the blank
- δ is the state transition function mapping $Q \times \Gamma$ to $Q \times \Gamma \times \{L, R\}$
- q_0 is the start state

- $F \subseteq Q$ is the set of final states.

A *configuration* of a Turing-machine M is written $\alpha q \beta$, where $q \in Q$ is the state of M in this configuration and $\alpha\beta$ is the relevant tape content with the head reading the next symbol to the right of α. This will be the leftmost symbol of β or a blank if β is empty. The *relevant tape content* extends right up to the first blank symbol or up to the head position if that position is further right. In the latter case the blank which is read by the head is not noted in the configuration. A configuration $X_1 \dots X_{i-2} p X_{i-1} Y X_{i+1} \dots X_n$ results from another configuration $X_1 \dots X_{i-1} q X_i \dots X_n$ by a *move* of M, written

$$X_1 \dots X_{i-1} q X_i \dots X_n \vdash_M X_1 \dots X_{i-2} p X_{i-1} Y X_{i+1} \dots X_n$$

if $\delta(q, X_i) = (p, Y, L)$, and

$$X_1 \dots X_{i-1} q X_i \dots X_n \vdash_M X_1 \dots X_{i-1} Y p X_{i+1} \dots X_n$$

if $\delta(q, X_i) = (p, Y, R)$, where $X_i = B$ if $i - 1 = n$.
A *computation* of a Turing-machine M is then a sequence of configurations in which successive configurations are $\vdash_M$-related. A *start configuration* of M for some input w is a configuration in which the head is positioned on the leftmost tape cell, the tape contains only w starting at the left and M is in state q_0. Every configuration in which M is in a state $q \in F$ is a *final configuration. M accepts an input w* if there is a computation of M beginning in the start configuration for w and ending in a final configuration. The *language accepted* by M, written $L(M)$, is then the set of inputs accepted by M.

Next we encode computations of Turing-machines as demands. To represent a computation the original input, the state reached and the relevant tape content left and right of the head suffice:

- for every state $q \in Q$ we use a 3-ary constructor $\mathtt{c}_q$,
- for every tape symbol $g \in \Gamma$ we use a 1-ary constructor $\mathtt{c}_g$ and
- we introduce a 0-ary constructor $\mathtt{c}_\varepsilon$ to terminate sequences of 1-ary constructors, i.e. representations of tape contents.

Using these constructors we are able to encode the tape content by representing a symbol followed by a possibly empty string of symbols as the application of an appropriate 1-ary constructor to the encoding of the remaining string of symbols. The empty string is encoded with the constructor $\mathtt{c}_\varepsilon$.

Computations are represented by the 3-ary constructors $\mathtt{c}_q$ for $q \in Q$, i.e. the current state is represented by the appropriate constructor. The first argument represents the original input, and the second and third argument represent the tape content produced. The second argument encodes the string to the left of the head in reverse, i.e. the symbol closest to the head is represented by the outermost constructor.

This intuition is formalized in the definitions and lemmata below. Our first step is to provide a demand representing the words over an alphabet Σ.

In order to increase readability we leave out confusing parentheses. In particular, we leave out the parentheses around the argument of a 1-ary constructor encoding symbols from the tape alphabet and only put successive applications of such constructors in parentheses if that is needed to delimit an argument in an application of one of the 3-ary constructors.

Definition and Lemma 3.54. *Let $\Sigma = \{s_1, \ldots, s_n\}$ be an alphabet.*

We define the demand

$$D_{\Sigma^*} \overset{def}{=} \langle \mathtt{c}_\varepsilon, \mathtt{c}_{s_1}\ D_{\Sigma^*}, \ldots, \mathtt{c}_{s_n}\ D_{\Sigma^*} \rangle$$

Then $t \sqsubseteq_p D_{\Sigma^*} \iff t \equiv \mathtt{c}_{r_1} \ldots \mathtt{c}_{r_m}\ \mathtt{c}_\varepsilon \wedge r_1 \ldots r_m \in \Sigma^*$.

Proof.

$\Longrightarrow$: From lemma 3.24 and lemma 3.26 we obtain $\exists i : D_p \sqsubseteq_p^i D_{\Sigma^*}$. Without loss of generality let i be the least number of iterations for which this holds. If $i = 1$ then $t \sqsubseteq_p^0 \mathtt{c}_\varepsilon$ and so $t \equiv \mathtt{c}_\varepsilon$. If $i > 1$ then $\Lambda_p(t) \subseteq \eta_{\Delta^{i-1}(\mathbf{0})}(\mathtt{c}_{r_1}\ D_{\Sigma^*}) = \{t | t \Downarrow \mathtt{c}_{r_1}\ t' \wedge t' \in \eta_{\Delta^{i-1}(\mathbf{0})}(D_{\Sigma^*})\}$. With the induction hypothesis we conclude the statement.

$\Longleftarrow$: It is easily seen, also by induction, that $t \equiv \mathtt{c}_{r_1} \ldots \mathtt{c}_{r_m}\ \mathtt{c}_\varepsilon \wedge r_1 \ldots r_m \in \Sigma^* \implies t \sqsubseteq_p^{m+1} D_{\Sigma^*}$. □

We define a demand D_{q_0} encoding the start configurations of M, i.e. we define a demand primitively including the encoding of the start configuration of the Turing-machine M for some input word.

Definition and Lemma 3.55. *Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be a Turing-machine. We define*

$$D_{q_0} \overset{def}{=} \mathtt{c}_{q_0}\ x\ \mathtt{c}_\varepsilon\ x \ \mathtt{where}\ x = D_{\Sigma^*}.$$

Then $\mathtt{c}_{q_0}\ (\mathtt{c}_{r_1} \ldots \mathtt{c}_{r_n}\ \mathtt{c}_\varepsilon)\ \mathtt{c}_\varepsilon\ (\mathtt{c}_{s_1} \ldots \mathtt{c}_{s_m}\ \mathtt{c}_\varepsilon) \sqsubseteq_p D_{q_0} \iff r_1 \ldots r_n = s_1 \ldots s_m \in \Sigma^*$.

Proof.

$\Longleftarrow$: If $r_1 \ldots r_n = s_1 \ldots s_m \in \Sigma^*$, then we know from the proof of lemma 3.54, that there is a least $i \geq 1$ for which

$\Lambda_p(\mathtt{c}_{r_1} \dots \mathtt{c}_{r_n}\ \mathtt{c}_\varepsilon) \subseteq \eta_{\Delta^i(\mathbf{0})}(D_{\Sigma^*}) = \Delta^i(\mathbf{0})(D_{\Sigma^*})$. It follows that

$$\begin{aligned}
&\Lambda_p(\mathtt{c}_{q_0}\ (\mathtt{c}_{r_1} \dots \mathtt{c}_{r_n}\ \mathtt{c}_\varepsilon)\ \mathtt{c}_\varepsilon\ (\mathtt{c}_{s_1} \dots \mathtt{c}_{s_m}\ \mathtt{c}_\varepsilon))\\
\subseteq&\eta_{\Delta^i(\mathbf{0})}(\mathtt{c}_{q_0}\ (\mathtt{c}_{r_1} \dots \mathtt{c}_{r_n}\ \mathtt{c}_\varepsilon)\ \mathtt{c}_\varepsilon\ (\mathtt{c}_{s_1} \dots \mathtt{c}_{s_m}\ \mathtt{c}_\varepsilon))\\
\subseteq&\bigcup_{\sigma\in\Sigma(x):\sigma x \sqsubseteq_p^i D_{\Sigma^*}} \eta_{\Delta^i(\mathbf{0})}(\sigma(\mathtt{c}_{q_0}\ x\ \mathtt{c}_\varepsilon\ x))\\
=&\eta_{\Delta^i(\mathbf{0})}(\mathtt{c}_{q_0}\ x\ \mathtt{c}_\varepsilon\ x \ \mathtt{where}\ x = D_{\Sigma^*})\\
=&\eta_{\Delta^{i+1}(\mathbf{0})}(D_{q_0})\\
\Longleftrightarrow&\ \mathtt{c}_{q_0}\ (\mathtt{c}_{r_1} \dots \mathtt{c}_{r_n}\ \mathtt{c}_\varepsilon)\ \mathtt{c}_\varepsilon\ (\mathtt{c}_{s_1} \dots \mathtt{c}_{s_m}\ \mathtt{c}_\varepsilon) \sqsubseteq_p^{i+1} D_{q_0}
\end{aligned}$$

$\neg \Longleftarrow \neg$: If $r_1 \dots r_n \neq s_1 \dots s_m$, then $n \neq m$ or $\exists i : r_i \neq s_i$. In both cases we can find a position at which $\mathtt{c}_{r_1} \dots \mathtt{c}_{r_n}\ \mathtt{c}_\varepsilon$ and $\mathtt{c}_{s_1} \dots \mathtt{c}_{s_m}\ \mathtt{c}_\varepsilon$ differ. For $m < n$ this is e.g. $\mathtt{c}_{r_{m+1}} \neq \mathtt{c}_\varepsilon$. From lemma 3.37 we see with induction that there cannot be a primitive demand C for which $\mathtt{c}_{r_1} \dots \mathtt{c}_{r_n}\ \mathtt{c}_\varepsilon \in \eta(C)$ as well as $\mathtt{c}_{s_1} \dots \mathtt{c}_{s_m}\ \mathtt{c}_\varepsilon \in \eta(C)$. This implies for all i:

$$\begin{aligned}
&\eta_{\Delta^i(\mathbf{0})}(\mathtt{c}_{q_0}\ (\mathtt{c}_{r_1} \dots \mathtt{c}_{r_n}\ \mathtt{c}_\varepsilon)\ \mathtt{c}_\varepsilon\ (\mathtt{c}_{s_1} \dots \mathtt{c}_{s_m}\ \mathtt{c}_\varepsilon))\\
&\qquad\cap \bigcup_{\sigma\in\Sigma(x):\sigma x \sqsubseteq_p^i D_{\Sigma^*}} \eta_{\Delta^i(\mathbf{0})}(\sigma(\mathtt{c}_{q_0}\ x\ \mathtt{c}_\varepsilon\ x)) = \emptyset
\end{aligned}$$

But since by definition 3.18 $\forall \rho$: $\Lambda_p(\mathtt{c}_{q_0}\ (\mathtt{c}_{r_1} \dots \mathtt{c}_{r_n}\ \mathtt{c}_\varepsilon)\ \mathtt{c}_\varepsilon\ (\mathtt{c}_{s_1} \dots \mathtt{c}_{s_m}\ \mathtt{c}_\varepsilon)) \subseteq \eta_\rho(\mathtt{c}_{q_0}\ (\mathtt{c}_{r_1} \dots \mathtt{c}_{r_n}\ \mathtt{c}_\varepsilon)\ \mathtt{c}_\varepsilon\ (\mathtt{c}_{s_1} \dots \mathtt{c}_{s_m}\ \mathtt{c}_\varepsilon))$ our claim must hold. □

We proceed to define the demand $D_{\vdash_M}$, the demand containing exactly the normal forms encoding computations of M.
The parameterized demands $D_{\vdash_M}(\delta(q,s) = (p,g,L))$ and $D_{\vdash_M}(\delta(q,s) = (p,g,R))$ (cf. definition 3.56) are defined using mu-

tual recursion via $D_{\vdash_M}$ to produce expressions expanding the encoding of a computation by applying the move $\delta(q,s) = (p,g,L)$ or $\delta(q,s) = (p,g,R)$, respectively.
As an example, consider the definition of $D_{\vdash_M}(\delta(q,s) = (p,g,L))$ in definition 3.56. There the pattern is used to match exactly the encodings representing computations starting with some input x and leading to a state q with the symbol s under M's head. We use the different demands for the constructors $\mathtt{c}_{r_1}, \dots, \mathtt{c}_{r_n}$ moving a particular constructor from the front of the second argument to the front of the third argument to represent M's head movement to the left.

Definition 3.56. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, $\Gamma = \{B, r_1, \dots, r_n\}$ and let δ be defined by $\delta(q_1, s_1) = (p_1, g_1, M_1), \dots, \delta(q_m, s_m) = (p_m, g_m, M_m)$ where the M_i are head movements from $\{L, R\}$.

$$\begin{aligned}
&D_{\vdash_M}(\delta(q,s) = (p,g,L))\\
&\overset{\text{def}}{=}\langle \mathtt{c}_p\ x\ y\ (\mathtt{c}_{r_1}\ \mathtt{c}_g\ z)\ \mathtt{where}\ \mathtt{c}_q\ x\ (\mathtt{c}_{r_1}\ y)\ (\mathtt{c}_s\ z) = D_{\vdash_M},\\
&\qquad \dots, \mathtt{c}_p\ x\ y\ (\mathtt{c}_{r_n}\ \mathtt{c}_g\ z)\ \mathtt{where}\ \mathtt{c}_q\ x\ (\mathtt{c}_{r_n}\ y)\ (\mathtt{c}_s\ z) = D_{\vdash_M}\rangle
\end{aligned}$$

$$\begin{aligned}
&D_{\vdash_M}(\delta(q,s) = (p,g,R))\\
&\overset{\text{def}}{=}\ \langle \mathtt{c}_p\ x\ (\mathtt{c}_g\ y)\ z\ \mathtt{where}\ \mathtt{c}_q\ x\ y\ (\mathtt{c}_s\ z) = D_{\vdash_M},\\
&\qquad \mathtt{c}_p\ x\ (\mathtt{c}_g\ y)\ (\mathtt{c}_B\ \mathtt{c}_\varepsilon)\ \mathtt{where}\ \mathtt{c}_q\ x\ y\ (\mathtt{c}_s\ \mathtt{c}_\varepsilon) = D_{\vdash_M}\rangle
\end{aligned}$$

We can use this to define

$$\begin{aligned}
D_{\vdash_M} \overset{\text{def}}{=} &\langle D_{q_0}, D_{\vdash_M}(\delta(q_1,s_1) = (p_1,g_1,M_1)),\\
&\dots, D_{\vdash_M}(\delta(q_m,s_m) = (p_m,g_m,M_m))\rangle.
\end{aligned}$$

The following lemma states that $D_{\vdash_M}$ does indeed encode the computations of M. More precisely, it states that M reaches a state where the head is within the relevant tape content iff an appropriate encoding with the relevant tape content to the right of the head is represented by $D_{\vdash_M}$, and that M reaches a state in which the head is to the right of the relevant tape content iff an appropriate encoding with a single blank to the right of the head is represented by $D_{\vdash_M}$.

Lemma 3.57. *Let $x = x_1 \ldots x_n, y = y_1 \ldots y_m$.*

$$\begin{aligned} &\mathsf{c}_p\ (\mathsf{c}_{x_1} \ldots \mathsf{c}_{x_n}\ \mathsf{c}_\varepsilon)\ (\mathsf{c}_{y_{i-1}}\ \mathsf{c}_{y_{i-2}} \ldots \mathsf{c}_{y_1}\ \mathsf{c}_\varepsilon)\ (\mathsf{c}_{y_i} \ldots \mathsf{c}_{y_m}\ \mathsf{c}_\varepsilon) \\ &\in \eta(D_{\vdash_M}) \\ &\qquad\qquad \iff q_0 x \vdash_M^* y_1 \ldots y_{i-1} p y_i \ldots y_m \wedge m \geq i \end{aligned}$$

and

$$\begin{aligned} &\mathsf{c}_p\ (\mathsf{c}_{x_1} \ldots \mathsf{c}_{x_n}\ \mathsf{c}_\varepsilon)\ (\mathsf{c}_{y_m}\ \mathsf{c}_{y_{m-1}} \ldots \mathsf{c}_{y_1}\ \mathsf{c}_\varepsilon)\ (\mathsf{c}_B\ \mathsf{c}_\varepsilon) \in \eta(D_{\vdash_M}) \\ &\qquad\qquad \iff q_0 x \vdash_M^* y_1 \ldots y_m p \end{aligned}$$

Proof. The proof is by induction on the length of M's computation. Lemma 3.55 contributes the base case. If $q_0 x \vdash_M^* y_1 \ldots y_{i-1} p y_i \ldots y_m$, $m \geq i-1$ is a computation of length $n > 0$, then there are different possibilities for the last move in this computation.

1. The head is positioned to the right of the relevant tape content in the configuration before the last move. This move is either
   a) a move to the left, $y_1 \ldots y_i p' \vdash_M y_1 \ldots y_{i-1} p y_i \ldots y_m$ where $m = i-1$ if $y - i = B$ and a B is written, or

   b) a move to the right, $y_1 \ldots y_{i-2} p' \vdash_M y_1 \ldots y_{i-1} p$.

From the induction hypothesis we obtain for the first case

$$\begin{aligned} &q_0 x \vdash_M^* y_1 \ldots y_{i-1} p' \\ \iff &\mathsf{c}_{p'}\ (\mathsf{c}_{x_1} \ldots \mathsf{c}_{x_n}\ \mathsf{c}_\varepsilon)\ (\mathsf{c}_{y_i} \ldots \mathsf{c}_{y_1}\ \mathsf{c}_\varepsilon)\ \mathsf{c}_\varepsilon \in \eta(D_{\vdash_M}) \\ \iff &\exists j : \mathsf{c}_{p'}\ (\mathsf{c}_{x_1} \ldots \mathsf{c}_{x_n}\ \mathsf{c}_\varepsilon)\ (\mathsf{c}_{y_i} \ldots \mathsf{c}_{y_1}\ \mathsf{c}_\varepsilon)\ \mathsf{c}_\varepsilon \\ &\in \eta_{\Delta^j(\mathbf{0})}(D_{\vdash_M}) \end{aligned}$$

and for the second

$$\begin{aligned} &q_0 x \vdash_M^* y_1 \ldots y_{i-2} p' \\ \iff &\mathsf{c}_{p'}\ (\mathsf{c}_{x_1} \ldots \mathsf{c}_{x_n}\ \mathsf{c}_\varepsilon)\ (\mathsf{c}_{y_{i-2}} \ldots \mathsf{c}_{y_1}\ \mathsf{c}_\varepsilon)\ (\mathsf{c}_B\ \mathsf{c}_\varepsilon) \\ &\quad \in \eta(D_{\vdash_M}) \\ \iff &\exists j : \mathsf{c}_{p'}\ (\mathsf{c}_{x_1} \ldots \mathsf{c}_{x_n}\ \mathsf{c}_\varepsilon)\ (\mathsf{c}_{y_{i-2}} \ldots \mathsf{c}_{y_1}\ \mathsf{c}_\varepsilon)\ (\mathsf{c}_B\ \mathsf{c}_\varepsilon) \\ &\qquad \in \eta_{\Delta^j(\mathbf{0})}(D_{\vdash_M}) \end{aligned} \tag{3.12}$$

In the latter case $\delta(p', B) = (q, y_{i-1}, R)$ must hold. With the substitution $\sigma = \{x \mapsto \mathsf{c}_{x_1} \ldots \mathsf{c}_{x_n}\ \mathsf{c}_\varepsilon, y \mapsto \mathsf{c}_{y_{i-2}} \ldots \mathsf{c}_{y_1}\ \mathsf{c}_\varepsilon\}$ and a j satisfying (3.12) we find that $\sigma(\mathsf{c}_{p'}\ x\ y\ (\mathsf{c}_B\ \mathsf{c}_\varepsilon)) \sqsubseteq_p^j D_{\vdash_M}$ and by definition of $D_{\vdash_M}(\delta(p', B) = (p, y_{i-1}, R))$ we see that $\sigma(\mathsf{c}_p\ x\ (\mathsf{c}_{y_{i-1}}\ y)\ (\mathsf{c}_B\ \mathsf{c}_\varepsilon)) \in \eta_{\Delta^{j+1}(\mathbf{0})}(D_{\vdash_M}(\delta(p', B) = (p, y_{i-1}, R))) \subseteq \eta_{\Delta^{j+2}(\mathbf{0})}(D_{\vdash_M})$.

Conversely, if $\sigma(\mathsf{c}_p\ x\ (\mathsf{c}_{y_{i-1}}\ y)\ (\mathsf{c}_B\ \mathsf{c})) \in \eta_{\Delta^{j+1}(\mathbf{0})}(D_{\vdash_M}(\delta(p', B) = (p, y_{i-1}, R))) \subseteq \eta_{\Delta^{j+2}(\mathbf{0})}(D_{\vdash_M})$ with $\sigma(\mathsf{c}_{p'}\ x\ y\ (\mathsf{c}_B\ \mathsf{c}_\varepsilon)) \sqsubseteq_p^j D_{\vdash_M}$ then this implies $y_1 \ldots y_{i-2} p' \vdash_M y_1 \ldots y_{i-1} p$.

The argument for the first case proceeds along the same lines.

2. The head's position before the last move is at least one symbol to the left of the end of the relevant tape content. Again, the move is either
   a) a move to the left, $y_1 \dots y_i p' s y_{i+2} \dots y_m \vdash_M y_1 \dots y_{i-1} p y_i \dots y_m$, or
   b) a move to the right, $y_1 \dots y_{i-2} p' s y_i \dots y_m \vdash_M y_1 \dots y_{i-1} p y_i \dots y_m$.

   Here also do we employ the induction hypothesis to obtain

$$\begin{aligned} & q_0 x \vdash_M^* y_1 \dots y_i p' s y_{i+2} \dots y_m \\ \iff & \mathtt{c}_{p'}\ (\mathtt{c}_{x_1} \dots \mathtt{c}_{x_n}\ \mathtt{c}_\varepsilon)\ (\mathtt{c}_{y_i} \dots \mathtt{c}_{y_1}\ \mathtt{c}_\varepsilon) \\ & \quad (\mathtt{c}_s\ \mathtt{c}_{y_{i+2}} \dots \mathtt{c}_{y_m}\ \mathtt{c}_\varepsilon) \in \eta(D_{\vdash_M}) \\ \iff & \exists j : \mathtt{c}_{p'}\ (\mathtt{c}_{x_1} \dots \mathtt{c}_{x_n}\ \mathtt{c}_\varepsilon)\ (\mathtt{c}_{y_i} \dots \mathtt{c}_{y_1}\ \mathtt{c}_\varepsilon) \\ & \qquad (\mathtt{c}_s\ \mathtt{c}_{y_{i+2}} \dots \mathtt{c}_{y_m}\ \mathtt{c}_\varepsilon) \in \eta_{\Delta^j(\mathbf{0})}(D_{\vdash_M}) \end{aligned}$$

   and appropriately

$$\begin{aligned} & q_0 x \vdash_M^* y_1 \dots y_{i-2} p' s y_i \dots y_m \\ \iff & \mathtt{c}_{p'}\ (\mathtt{c}_{x_1} \dots \mathtt{c}_{x_n}\ \mathtt{c}_\varepsilon)\ (\mathtt{c}_{y_{i-2}} \dots \mathtt{c}_{y_1}\ \mathtt{c}_\varepsilon) \\ & \quad (\mathtt{c}_s\ \mathtt{c}_{y_i} \dots \mathtt{c}_{y_m}\ \mathtt{c}_\varepsilon) \in \eta(D_{\vdash_M}) \\ \iff & \exists j : \mathtt{c}_{p'}\ (\mathtt{c}_{x_1} \dots \mathtt{c}_{x_n}\ \mathtt{c}_\varepsilon)\ (\mathtt{c}_{y_{i-2}} \dots \mathtt{c}_{y_1}\ \mathtt{c}_\varepsilon) \\ & \quad (\mathtt{c}_s\ \mathtt{c}_{y_i} \dots \mathtt{c}_{y_m}\ \mathtt{c}_\varepsilon) \in \eta_{\Delta^j(\mathbf{0})}(D_{\vdash_M}) \end{aligned}$$

   in both cases the rest of the proof proceeds as for the two implications for the last move in 1. □

Thus we have a demand $D_{\vdash_M}$ for which the primitively included demands encode exactly the computations of a Turing-machine M. Accordingly, the next step in our presentation will be to construct a demand for $L(M)$, which is achieved by "cutting out" the

input component from expressions encoding accepting computations.

Definition 3.58. Let M be a Turing-machine with accepting states $F = \{q_1, \dots, q_n\}$. We define the demand

$$A_M \overset{\text{def}}{=} \langle x \ \mathtt{where}\ \mathtt{c}_{q_1}\ x\ y\ z = D_{\vdash_M}, \dots, x \ \mathtt{where}\ \mathtt{c}_{q_n}\ x\ y\ z = D_{\vdash_M} \rangle.$$

Theorem 3.59. *Let $x = x_1 \dots x_m$, M be a Turing-machine with $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, then $\mathtt{c}_{x_1} \dots \mathtt{c}_{x_m}\ \mathtt{c}_\varepsilon \sqsubseteq_p A_M \iff q_0 x \vdash_M^* \alpha p \beta$ and $p \in F$.*

Proof. The proof is similar to the proofs of lemma 3.57, lemma 3.55 and lemma 3.54. □

With the reduction of Turing-machines to demands we have shown that representations of demands can express at least as much structure as computations of Turing-machines. Now we want to show that their expressibility is indeed equivalent. This is done by presenting a method for enumerating the demands primitively included in an arbitrary demand. To this end we define a mapping, $\xi(\cdot)$, similar to $\eta(\cdot)$ (cf. definition 3.21), but yielding a set of primitive demands instead of the primitive sets, which $\eta(\cdot)$ yields. Essentially, this amounts to replacing the base cases of the recursive definition 3.18.

Definition 3.60. Let $\rho : N(\mathcal{C}) \to \mathcal{P}(\Lambda_C)$ and let P be the set of patterns. We define $\xi_\rho : \Lambda_C \mapsto \mathcal{P}(\Lambda_C)$ as

$$\begin{aligned} \xi_\rho(\mathtt{Bot}) &\overset{\text{def}}{=} \{\mathtt{Bot}\} \\ \xi_\rho(\mathtt{Fun}) &\overset{\text{def}}{=} \{\mathtt{Fun}\} \\ \xi_\rho(\mathtt{c}\ C_1 \dots C_{\alpha(\mathtt{c})}) &\overset{\text{def}}{=} \{\mathtt{c}\ t_1 \dots t_{\alpha(\mathtt{c})} | t_i \in \xi_\rho(C_i)\} \end{aligned}$$

$$\xi_\rho(\langle C_1,\dots,C_n\rangle) \overset{\text{def}}{=} \bigcup_i \xi_\rho(C_i)$$
$$\xi_\rho(C_1 \cap \dots \cap C_n) \overset{\text{def}}{=} \bigcap_i \xi_\rho(C_i)$$
$$\xi_\rho(N) \overset{\text{def}}{=} \rho(N),\ \text{if } N \in N(\mathcal{C})$$
$$\xi_\rho(S \texttt{ where } T = N) \overset{\text{def}}{=} \bigcup_{\sigma\in\Sigma(T):\sigma T\in\rho(N)} \xi_\rho(\sigma S).$$

The improvement of an approximation is obtained just as for η_ρ.

Definition 3.61. $\Delta'(\rho) \overset{\text{def}}{=} \rho'$, where $\rho'(N) = \xi_\rho(D)$ if $N = D \in \mathcal{C}$.

The proof of the prerequisites for applying the theorem of Knaster and Tarski works just as for Δ, so that we can define $\xi(C)$ with the least fixpoint of Δ'.

Definition 3.62. For $C \in \Lambda_C$: $\xi(C) \overset{\text{def}}{=} \xi_\beta(C)$ where $\beta = \mu x.\Delta'(x)$.

It is equally straightforward to see that the requirements for applying the CPO Fixpoint Theorem I [DP90] are met which proves the

Lemma 3.63. $\mu x.\Delta'(x) = \bigcup_{i\geq 0} \Delta'^i(\mathbf{0})$.

Now we can formulate the desired relation between η and ξ.

Proposition 3.64. *Let D_p be a primitive demand and let D be a demand, then*

$$D_p \sqsubseteq_p^i D \iff D_p \in \xi_{\Delta'^i(\mathbf{0})}(D).$$

Proof. Both implications are proved separately and we use induction on lexicographically ordered triples where the first component is the least number of iterations for which the requirement holds, the second is the number of `where`s in the demand D and the third is D itself. For the third component we use the structural order.

base case $(0, 0, D)$ where $D \equiv \texttt{Bot}$, $D \equiv \texttt{Fun}$ or $D \equiv \texttt{c}$ for a 0-ary constructor `c`. In these cases $D_p \equiv D$.

step Let (i, w, D) be the measure for the demand D and assume for all smaller (j, v, E) we know $D_p \sqsubseteq_p^j D \implies D_p \in \xi_{\Delta'^j(\mathbf{0})}(E)$. The following cases may occur.

$D \equiv \texttt{c}\ D_1 \dots D_{\alpha(\texttt{c})}$: Everyone of the triples (i, w_j, D_j) for $1 \leq j \leq \alpha(\texttt{c})$ is smaller than (i, w, D). $D_p \sqsubseteq_p^i D$ implies $D_p \equiv \texttt{c}\ D_1^p \dots D_{\alpha(\texttt{c})}^p$ and $\forall j : D_j^p \sqsubseteq_p^i D_j$. With the induction hypothesis we obtain $\forall j : D_j^p \in \xi_{\Delta'^i(\mathbf{0})}(D_j)$ which in turn implies $D_p \in \xi_{\Delta'^i(\mathbf{0})}(D)$.

$D \equiv \langle D_1, \dots, D_n\rangle$: Similar to $D \equiv \texttt{c}\ D_1 \dots D_{\alpha(\texttt{c})}$.

$D \equiv D_1 \cap \dots \cap D_n$: Similar to $D \equiv \texttt{c}\ D_1 \dots D_{\alpha(\texttt{c})}$.

$D \equiv N$: $\eta_{\Delta^i(\mathbf{0})}(N) = \Delta^i(\mathbf{0})(N) = \eta_{\Delta^{i-1}(\mathbf{0})}(E)$ if $N = E \in \mathcal{C}$ and accordingly for $\xi_{\Delta'^i(\mathbf{0})}$. Since $(i-1, v, E) < (i, w, D)$ the induction hypothesis yields $D_p \sqsubseteq_p^{i-1} E \implies D_p \in \xi_{\Delta^{i-1}(\mathbf{0})}(E)$ and thus $D_p \sqsubseteq_p^i D \implies D_p \in \xi_{\Delta^i(\mathbf{0})}(D)$.

$D \equiv S \texttt{ where } T = N$: $\eta_{\Delta^i(\mathbf{0})}(S \texttt{ where } T = N) = \bigcup_{\sigma\in\Sigma(T):\sigma T\sqsubseteq_p^i N} \eta_{\Delta^i(\mathbf{0})}(\sigma S)$ and due to lemma 3.26 there is a $\sigma \in \Sigma(T) : \sigma T \sqsubseteq_p^i N \wedge D_p \sqsubseteq_p^i \sigma S$. We can employ the induction hypothesis since $(i, w-1, \sigma S) < (i, w, D)$. Additionally, $\Delta^i(\mathbf{0})(N) = \eta_{\Delta^{i-1}(\mathbf{0})}(E)$ if $N = E \in \mathcal{C}$ and thus $\sigma T \sqsubseteq_p^i N \iff \sigma T \sqsubseteq_p^{i-1} E$ and again we can apply the induction hypothesis. We obtain for some $\sigma : \sigma T \in \xi_{\Delta'^{i-1}(\mathbf{0})}(E) \wedge D_p \in \xi_{\Delta'^{i}(\mathbf{0})}(\sigma S)$ and thus $D_p \in \bigcup_{\sigma \in \Sigma(T):\sigma T \in \Delta'^{i}(\mathbf{0})(N)} \xi_{\Delta'^{i}(\mathbf{0})}(\sigma S)$.

The remaining implication is proved analogously. □

The same induction proves the

Lemma 3.65. *Let $D \in \Lambda_C$, then $\xi_{\Delta'^{i}(\mathbf{0})}(D)$ is finite for every i.*

The procedure for enumerating all of the demands primitively included in a given demand D then amounts to enumerating the finitely many primitive demands for each of the iterations of Δ', i.e. $\xi_{\Delta'^{i}(\mathbf{0})}(D)$, in order of increasing i.
Emphasizing the result of this section, we state the

Theorem 3.66. *Demands and Turing-machines have the same expressibility.*

A consequence of the equivalence of Turing-machines and demands is that there cannot be a procedure to compute a demand representing the complement of a given demand. If there would be such a procedure we could decide the halting problem by encoding a Turing-machine's accepted language as a demand, complementing and enumerating the encoding demand as well as its complement. Other undecidable properties are accordingly transferred to demands.

Theorem 3.67. *Undecidable properties are:*

membership	$w \sqsubseteq_p D$
consistency	$C \neq \emptyset$
inclusion	$C \leq_\gamma D$.

3.5 Closure of club

We will now show that club is a closure operator. One motivation to do this is that γ is defined with an application of the club operator and that we want to conclude e.g. from $\mathbb{C}[\eta(D)] \subseteq \gamma(E)$ that $\mathbb{C}[\gamma(D)] \subseteq \gamma(E)$.
Thus we will not need closure of the club operator for arbitrary sets and we will show this property only for particular sets, the *primitive sets*. The representation of demands as well as their concretization are primitive sets.
It is possible that club-closure could be proved for more general sets, i.e. sets which are not primitive sets. In the proofs below we do make use of the properties of primitive sets, in such a way that a proof of the club-closure of non-primitive sets would need a different proof method.
Subsequently, we will construct from a so called *double chain* s_{ij} (cf. definition 3.76) with $\bigsqcup_i^c \bigsqcup_j^c s_{ij} \equiv_c t$ a chain s^i with $\bigsqcup_i^c s^i \equiv_c t$. The construction proceeds by choosing appropriate chain elements from the s_{ij} and, in case of sub-terms with FWHNFs, replacing sub-terms of the s_{ij} with appropriate sub-terms of t. Diagonalization alone is *not* sufficient in our setting, instead it is necessary to replace the sub-terms with FWHNFs. The reason for this will shortly become clear. The diagram of figure 3.2 illustrates the construction.
Here the mapping from e.g. $s_{11} \mapsto s^1$ represents the possible replacement of a sub-term with FWHNF. Intuitively, the edges from s^i to s^{i+1} indicate that s^{i+1} is greater than s^i at all positions where s^i differs from t. We will now formally introduce the notions "position", "greater", etc.
Our objective to show that club is a closure operator needs no

$$\begin{tikzcd}
t_1 & t_2 & t_3 & t_4 & \cdots \arrow[r, "\bigsqcup^c"] & t \\
\vdots \arrow[u, "\bigsqcup^c"'] & \vdots \arrow[u, "\bigsqcup^c"'] & \vdots \arrow[u, "\bigsqcup^c"'] & \vdots \arrow[u, "\bigsqcup^c"'] & & \\
s_{14} & s_{24} & s_{34} \mapsto s^3 \arrow[ur] & s_{44} & & \\
s_{13} & s_{23} \mapsto s^2 \arrow[ur] & s_{33} & s_{43} & & \\
s_{12} & s_{22} & s_{32} & s_{42} & & \\
s_{11} \mapsto s^1 \arrow[uur] & s_{21} & s_{31} & s_{41} & &
\end{tikzcd}$$

Figure 3.2: Construction of a chain from a double chain

proof if t or all the t_i are primitive. Non-primitive expressions t will reduce to an expression with unbounded constructor nesting.

Definition 3.68 (primitive position). Let s be a Λ-expression. The *primitive positions* of s, $\Pi(s)$, are defined as the set satisfying

1. the empty word, λ, is a primitive position of s, and
2. if $s\Downarrow_S \mathsf{c}\ s_1 \dots s_{\alpha(\mathsf{c})}$, $\imath \in \{1, \dots, \alpha(\mathsf{c})\}$ and π is a primitive position of $s_\imath$, then $\mathsf{c} \cdot \imath \cdot \pi$ is a primitive position of s.

For a primitive position π of s we define

$$s_{|\pi} \overset{\text{def}}{=} s, \text{ if } \pi = \lambda \text{ and}$$
$$s_{|\pi} \overset{\text{def}}{=} s_{\imath|\rho}, \text{ if } s\Downarrow_S \mathsf{c}\ s_1 \dots s_{\alpha(\mathsf{c})} \text{ and } \pi = \mathsf{c} \cdot \imath \cdot \rho$$

Analogously, we define

$$s[\pi \mapsto t] \overset{\text{def}}{=} t, \text{ if } \pi = \lambda \text{ and}$$
$$s[\pi \mapsto t] \overset{\text{def}}{=} s_{|\imath}[\rho \mapsto t], \text{ if } s\Downarrow_S \mathsf{c}\ s_1 \dots s_{\alpha(\mathsf{c})} \text{ and } \pi = \mathsf{c} \cdot \imath \cdot \rho$$

A primitive position π of s is called *maximal primitive position* of s, iff the only primitive position of $s_{|\pi}$ is λ.

The notion of primitive position should not be confused with that of position, apart from the constructor label the primitive position refers to a position *in the SCWHNF that expression reduces to* whereas a position of an expression only refers to the syntax of an expression and does not involve any reduction. If we want to emphasize the difference between positions and primitive positions we use the term *manifest position* for the former.

Remark 3.69. Any primitive position can be regarded as a position in a sufficiently reduced expression by ignoring the constructors in it.

With definition 3.68 we can characterize the primitive expressions as those having finitely many primitive positions. As a tool used in this characterization we introduce a partial order on sets of primitive positions on which we can base inductions, if these sets are finite.

Definition 3.70. Let Ψ and Π be possibly infinite sets of primitive positions of some Λ-expressions s and t, respectively. $\Psi \leq \Pi$ iff there is a primitive position π, such that $\pi \cdot \Psi \subseteq \Pi$. As usual we write $\Psi < \Pi$ if $\Psi \leq \Pi$, but $\Psi \not\equiv \Pi$.

Lemma 3.71. *$\leq$ on sets of primitive positions is a partial order.*

Proof. Obviously, $\Psi \leq \Psi$, $\Psi \leq \Pi \wedge \Pi \leq \Psi \iff \Psi = \Pi$ and $\Psi \leq \Pi \wedge \Pi \leq \Phi \implies \Psi \leq \Phi$. □

Lemma 3.72. *An expression $s \in \Lambda$ is primitive iff $\Pi(s)$ is finite and $\forall \pi \in \Pi(s) : s_{|\pi} \overset{*}{\not\to} \mathbb{R}[x]$.*

Proof.

$\Longleftarrow$: We use induction on $\Pi(s)$ with the partial order from definition 3.70.

$\Pi(s) = \{\lambda\}$: One of $s\Uparrow$ or $s\Downarrow_F$ must hold. In the first case $s \in \eta(\texttt{Bot})$, in the second $s \in \eta(\texttt{Fun})$.

$\Pi(s) \supset \{\lambda\}$: $s\Downarrow_S \texttt{c}\ s_1 \dots s_{\alpha(\texttt{c})}$ must hold. Then $\texttt{c} \cdot \imath \cdot \Pi(s_\imath) \subseteq \Pi(s)$ and thus $\Pi(s_\imath) < \Pi(s)$. From the induction hypothesis we conclude that the constructor arguments are primitive and thus s is primitive.

$\Longrightarrow$: If s is primitive then there is a demand D with $s \in \Lambda_p(D)$. D has finite depth and consists of constructors, `Bot` and `Fun`. It is easily seen that s can only have finitely many primitive positions. □

Corollary 3.73. *A closed expression $s \in \Lambda$ is primitive iff $\Pi(s)$ is finite.*

Corollary 3.74. *A closed expression $s \in \Lambda$ is non-primitive iff s has arbitrarily long primitive positions.*

The sets for which we will show that the club operation does not add anything to their club will have to satisfy some conditions: they will need to be *primitive sets*.

Definition 3.75 (primitive set). Let $A \subseteq \Lambda$. A is called *primitive set*, iff A is the union of the $\equiv_c$-equivalence classes of some closed primitive expressions $s \in \Lambda_p$, and A satisfies $\forall s \in A \cap \Lambda_p, \pi \in \Pi(s) : s_{|\pi}\Downarrow_F \implies (\forall t \in \Lambda : t\Downarrow_F \implies s[\pi \mapsto t] \in A)$.

Thus the primitive sets must be closed with respect to $\equiv_c$ and with respect to replacing subexpressions having an FWHNF with other FWHNFs.
Obviously, speaking of subexpression is a slight abuse of definition, but since the $\equiv_c$-equivalence classes are closed with respect to $\to$ it suffices to speak about subexpressions and have other appropriate expressions included into the primitive set by the $\equiv_c$-closure.

Definition 3.76 (double chain). $s_{ij} \in \Lambda$, $i, j \in \mathbb{N}_0$ is called a *double chain* for t, iff

$$\begin{aligned}
&(\forall i, j : s_{ij} \leq_c s_{ij+1}) \\
\wedge&(\forall i \exists t_i \in \Lambda : \bigsqcup_j^c s_{ij} \equiv_c t_i) \\
\wedge&(\forall i : t_i \leq_c t_{i+1}) \wedge \bigsqcup_i^c t_i \equiv_c t.
\end{aligned}$$

Our aim is to construct an ascending chain s^i from the double chain with the same club, t, by choosing appropriate elements among the s_{ij} and substituting some subexpressions of these expressions. For this choice it is not sufficient to ensure that the s^i are contextually ordered. Intuitively, the reason for this insufficiency is that, although contextually ordered, the s^i may "grow in one dimension only" whereas the s_{ij} could "grow in more than one dimension". The following example will illustrate this.

Example 3.77. $t \equiv_c$ `repeat (repeat 1)`, $t_i \equiv_c$ `repeat ((1 :)`^{i} `bot)` and $s_{ij} \equiv_c$ `(((1 :)`^{i} `bot) :)`^{j} `bot`. We define $s^k \overset{\text{def}}{=} s_{kn}$, for some n, and satisfy $s^i <_c s^{i+1}$, but $\bigsqcup_i^c s^i \equiv_c$ `((repeat 1) :)`^{n} `bot` $\not\equiv_c t$. Intuitively, the s_{ij} grow along two "dimensions", but s^k grows only along one (see figure 3.3).

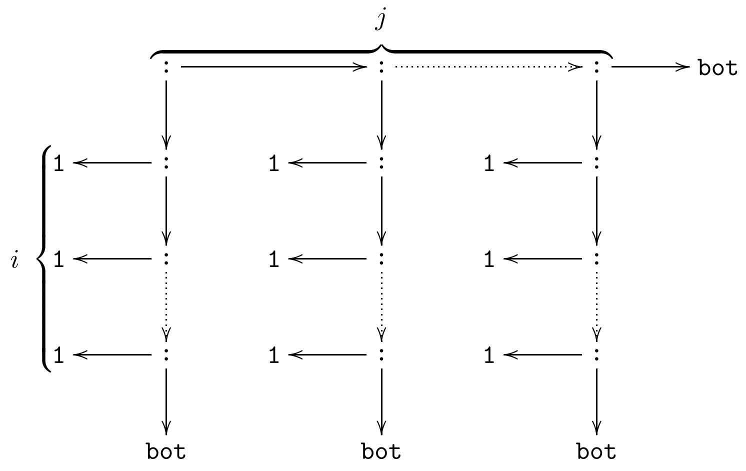


Figure 3.3: Double chain for `repeat (repeat 1)`

This diagram represents the elements of the double chain s_{ij}. The root of the graph is in the upper left corner. Every node labeled `:` represents an application of the `:`-constructor. The two arguments of this constructor are connected to such a node by outgoing edges counterclockwise from left to right.

We see that the strict contextual order is not a sufficient criterion for the chosen chain to have the intended club. Instead we need

to ensure that progress is made along all the dimensions along which the s_{ij} grow, and we form the notion of "point-wise" contextual order and the related notion of differing positions. Our need to ensure growth along all dimensions is also the reason for replacing subexpressions having an FWHNF with the appropriate subexpression of t. Otherwise we would need entirely different definitions for primitive positions, $\leq_\pi$ etc. to state that applications of these FWHNFs would grow along all of their "dimensions". Later we will see that if our chain s^i grows along all the differing positions then it does indeed have the intended club.

Definition 3.78 ($\leq_\pi$). Let s, t be Λ-expressions and $\pi \in \Pi(s)$. $s \leq_\pi t$ iff

$$\pi \in \Pi(t) \tag{3.13}$$

$$\text{and } s_{|\pi} \leq_c t_{|\pi}. \tag{3.14}$$

Definition 3.79 (differing position). Let s, t be Λ-expressions, $s \leq_c t$ and let $\pi \in \Pi(s)$. π is a *differing position* of s and t, iff $s <_\pi t$.

Differing positions need not be maximal primitive positions, so prefixes of differing positions will be differing positions and differing positions may also be prefixes of primitive positions that are not differing.
For every Λ-expression s and every primitive position we define the function $f_{s,\pi}$, such that the application of $f_{s,\pi}$ to s selects the subexpression of s at the primitive position π. If the argument of $f_{s,\pi}$ does not have the primitive position π, then the application has no WHNF. Furthermore $f_{s,\pi}\ s \leq_c f_{s,\pi}\ t$ iff $s \leq_\pi t$ provided $\pi \in \Pi(t)$. (The identity of the subscript s and the argument s on the left of $\leq_c$ intended.)

Definition 3.80 ($f_{s,\pi}$). Let s be a Λ-expression.

$$f_{s,\lambda} \overset{\text{def}}{=} \lambda x.x$$

$$f_{s,\mathtt{c}_{A,k}\cdot\imath\cdot\pi} \overset{\text{def}}{=} \lambda x.f_{s_\imath,\pi}\ (\mathtt{case}\ x\ \overbrace{\mathtt{bot}\ldots\mathtt{bot}}^{k-1}\ (\lambda x_1\ldots x_m.x_\imath)\ \underbrace{\mathtt{bot}\ldots\mathtt{bot}}_{k-1}),\ \text{if } s{\Downarrow_S}\mathtt{c}_{A,k}\ s_1\ldots s_m.$$

In lemma 3.81 we prove that $f_{s,\pi}$ is strict. This will make inductive arguments easier, since it will allow to focus on the result of $f_{s_\imath,\pi}$'s argument in applications such as $f_{s_\imath,\pi}\ (\mathtt{case}_A\ t\ \underbrace{\mathtt{bot}\ldots\mathtt{bot}}_{k-1}\ (\lambda x_1\ldots x_m.x_\imath)\ \mathtt{bot}\ldots\mathtt{bot})$.

Lemma 3.81. *Let s be a Λ-expression and $\pi\in\Pi(s)$. Then $f_{s,\pi}$ is strict.*

Proof. We use induction on π.

$\pi=\lambda:$ From $t{\Uparrow}$ we see that $f_{s,\lambda}\ t{\Uparrow}$, since $f_{s,\lambda}\ t=(\lambda x.x)\ t\to_{\text{no}} t$.

$\pi=\mathtt{c}_{A,k}\cdot\imath\cdot\rho:$ $f_{s,\mathtt{c}_{A,k}\cdot\imath\cdot\rho}\ t \xrightarrow{*}_{\text{no}} f_{s_\imath,\rho}(\mathtt{case}_A\ t\ \underbrace{\mathtt{bot}\ldots\mathtt{bot}}_{k-1}\ (\lambda\vec{x}.x_\imath)\ \mathtt{bot}\ldots\mathtt{bot})$ and by induction hypothesis $f_{s_\imath,\rho}$ is strict. Since `case` is strict in its first argument $t{\Uparrow} \implies f_{s,\pi}\ t{\Uparrow}$ and $f_{s,\pi}$ is strict. □

Lemma 3.82. *Let s,t be Λ-expressions and let π be a primitive position of s.*

1. *If t has the primitive position π, then $f_{s,\pi}\ t\xrightarrow{*}_{no} t_{|\pi}$.*
2. *If t does not have the primitive position π, then $f_{s,\pi}\ t$ has no WHNF*

Proof. We show both statements by induction on the primitive position π.

1\.

$\pi=\lambda:$ $f_{s,\lambda}\ t=(\lambda x.x)\ t\to_{\text{no}} t$ and $t=t_{|\lambda}$.

$\pi=\mathtt{c}_{A,k}\cdot\imath\cdot\rho:$ $f_{s,\pi}\ s \xrightarrow{*}_{\text{no}} f_{s_\imath,\rho}\ (\mathtt{case}_A\ t\ \underbrace{\mathtt{bot}\ldots\mathtt{bot}}_{k-1}\ (\lambda\vec{x}.x_\imath)\ \mathtt{bot}\ldots\mathtt{bot})$. Since t has the primitive position π, $t{\Downarrow_S}\mathtt{c}_{A,k}\ t_1\ldots t_{\alpha(\mathtt{c}_{A,k})}$. Because $f_{s_\imath,\rho}$ is strict, the argument will definitely be evaluated in an evaluation to WHNF and $\mathtt{case}_A\ t\ \underbrace{\mathtt{bot}\ldots\mathtt{bot}}_{k-1}\ (\lambda x_1\ldots x_m.x_\imath)\ \mathtt{bot}\ldots\mathtt{bot}\xrightarrow{*}_{\text{no}} t_\imath$. The induction hypothesis implies that $f_{s_\imath,\rho}\ t_\imath$ reduces to $t_{\imath|\rho}$. Then $f_{s,\pi}\ t$ reduces to $t_{|\pi}$.

2\.

$\pi=\lambda:$ Since every expression t has the primitive position λ, the statement holds.

$\pi=\mathtt{c}_{A,k}\cdot\imath\cdot\rho:$ $f_{s,\pi}\ t \to_{\text{no}} f_{s_\imath,\rho}\ (\mathtt{case}_A\ t\ \underbrace{\mathtt{bot}\ldots\mathtt{bot}}_{k-1}\ (\lambda\vec{x}.x_\imath)\ \mathtt{bot}\ldots\mathtt{bot})$. t does not have the primitive position $\mathtt{c}_{A,k}\cdot\imath\cdot\rho$, if t does not have the primitive position $\mathtt{c}_{A,k}\cdot\imath$ or t has a subexpression t' at primitive position $\mathtt{c}_{A,k}\cdot\imath$, but ρ is not a primitive position in t'. In the first case $t{\not\Downarrow_S}\mathtt{c}_{A,k}\ t_1\ldots t_m$. So $t{\Uparrow}$, $t{\Downarrow_F}$, $t\xrightarrow{*}\mathbb{R}[x]$ or $t{\Downarrow_S}\mathtt{c}_{B,j}\ t'_1\ldots t'_{m'}$ with $A\neq B$ or $i\neq j$. In all these cases it is easily seen that

$$(\mathtt{case}_A\ t\ \underbrace{\mathtt{bot}\ldots\mathtt{bot}}_{k-1}\ (\lambda x_1\ldots x_m.x_\imath)\ \mathtt{bot}\ldots\mathtt{bot}){\Uparrow}$$

and thus

$$f_{s_\imath,\rho}\,(\mathtt{case}_A\ t\ \underbrace{\mathtt{bot}\ldots\mathtt{bot}}_{k-1}\,(\lambda x_1\ldots x_m.x_\imath)\ \mathtt{bot}\ldots\mathtt{bot})\Uparrow$$

or that

$$\mathtt{case}_A\ \mathbb{R}[x]\ \underbrace{\mathtt{bot}\ldots\mathtt{bot}}_{k-1}\,(\lambda x_1\ldots x_m.x_\imath)\ \mathtt{bot}\ldots\mathtt{bot}\not\to_{\mathrm{no}}$$

and thus

$$f_{s_\imath,\rho}\ (\ldots)\not\Downarrow.$$

In the second case we also deduce from the induction hypothesis that $f_{s_\imath,\rho}\ t'\Uparrow$. □

Lemma 3.83. *Let s, t be Λ-expressions.*

$$s\le_c t \implies \Pi(s)\subseteq\Pi(t).$$

Proof. Assume this is false. Then there are s, t satisfying

$$s\le_c t\wedge\exists\pi\in\Pi(s):\pi\notin\Pi(t) \tag{3.15}$$

$\pi=\lambda$ obviously cannot satisfy (3.15). Assuming π to be the smallest primitive position satisfying (3.15) now implies $\pi=\mathtt{c}\cdot\imath\cdot\rho$ and $s\Downarrow_S\mathtt{c}\ s_1\ldots s_m$. From lemma 2.112 we obtain $t\Downarrow_S\mathtt{c}\ t_1\ldots t_m$ and lemma 2.95 implies $\forall 1\le\imath\le\alpha(\mathtt{c}):s_\imath\le_c t_\imath$. With the induction hypothesis we obtain the statement. □

Lemma 3.84. *Let s, t be Λ-expressions and $\pi_1\cdot\pi_2$ be a primitive position of s with $s_{|\pi_1\cdot\pi_2}\not\equiv_c\mathtt{bot}$. Then $f_{s,\pi_1}\ s\le_{\pi_2} f_{s,\pi_1}\ t$ iff $s\le_{\pi_1\cdot\pi_2} t$.*

Proof. We use induction on π_1.

$\pi_1=\lambda$:

$$\begin{aligned}
&f_{s,\pi_1}\ s\le_{\pi_2} f_{s,\pi_1}\ t\\
\iff&(f_{s,\pi_1}\ s=(\lambda x.x)\ s\to_{\mathrm{no}} s\text{ and }f_{s,\pi_1}\ t\to_{\mathrm{no}} t)\\
&s\le_{\pi_2} t\\
\iff&(\pi_1\cdot\pi_2=\pi_2)\\
&s\le_{\pi_1\cdot\pi_2} t
\end{aligned}$$

$\pi_1=\mathtt{c}_{A,k}\cdot\imath\cdot\rho$:

$$\begin{aligned}
&f_{s,\pi_1}\ s\le_{\pi_2} f_{s,\pi_1}\ t\\
\iff&(\text{definition 3.80}+\xrightarrow{*}_{\mathrm{no}})\\
&f_{s_\imath,\rho}\ (\mathtt{case}_A\ s\ \underbrace{\mathtt{bot}\ldots\mathtt{bot}}_{k-1}\ (\lambda x_1\ldots x_m.x_\imath)\ \vec{\mathtt{bot}})\\
&\quad\le_{\pi_2} f_{s_\imath,\rho}\ (\mathtt{case}_A\ t\ \ldots)\\
\iff&(s\Downarrow_S\mathtt{c}_{A,k}\ s_1\ldots s_{\alpha(\mathtt{c}_{A,k})}\text{ and }t\Downarrow_S\mathtt{c}_{A,k}\ t_1\ldots t_{\alpha(\mathtt{c}_{A,k})}\\
&\text{since }\pi_1\cdot\pi_2\in\Pi(s)\wedge s_{|\pi_1\cdot\pi_2}\not\equiv_c\mathtt{bot}+\xrightarrow{*}_{\mathrm{no}})\\
&f_{s_\imath,\rho}\ s_\imath\le_{\pi_2} f_{s_\imath,\rho}\ t_\imath\\
\iff&(\text{induction hypothesis})\\
&s_\imath\le_{\rho\cdot\pi_2} t_\imath\\
\iff&(\text{induction hypothesis})\\
&s\le_{\mathtt{c}_{A,k}\cdot\imath\cdot\rho\cdot\pi_2} t\\
\iff&\\
&s\le_{\pi_1\cdot\pi_2} t\quad\square
\end{aligned}$$

Corollary 3.85. *Let s, t be Λ-expressions and $\pi\in\Pi(s)\cap\Pi(t)$. Then*

$$f_{s,\pi}\ s\le_c f_{s,\pi}\ t\iff s\le_\pi t.$$

Proof.

$$
\begin{aligned}
& f_{s,\pi}\ s \leq_c f_{s,\pi}\ t \\
\Longleftrightarrow\ & \text{(every expression has the primitive position } \lambda, \\
& \ \lambda \text{ has no proper prefix)} \\
& f_{s,\pi}\ s \leq_\lambda f_{s,\pi}\ t \\
\Longleftrightarrow\ & \text{(lemma 3.84)} \\
& s \leq_{\pi\cdot\lambda} t \\
\Longleftrightarrow\ & \\
& s \leq_\pi t. \quad \square
\end{aligned}
$$

The following lemma proves the $\leq_\pi$ relation to be a refinement of the $\leq_c$ relation. This will later be quite useful in contradiction proofs, since assuming $s \not\leq_c t$ allows us to focus on a primitive position π for which $s \leq_\pi t$

Lemma 3.86. *Let s, t be Λ-expressions.*

$$s \leq_c t \iff \forall \pi \in \Pi(s) : s \leq_\pi t.$$

Proof.

$\Longrightarrow$: With lemma 3.83 we obtain $\Pi(s) \subseteq \Pi(t)$, satisfying (3.13). Assume $s \leq_c t$ and $\exists \pi \in \Pi(s) : s_{|\pi} \not\leq_c t_{|\pi}$. Then there is a context $\mathbb{C}[\cdot]$ with $\mathbb{C}[s_{|\pi}]\Downarrow$, but $\mathbb{C}[t_{|\pi}]\not\Downarrow$. With $\mathbb{D}[\cdot] \overset{\text{def}}{=} f_{s,\pi}\ [\cdot]$ we obtain $\mathbb{C}[\mathbb{D}[s]]\Downarrow$ and $\mathbb{C}[\mathbb{D}[t]]\not\Downarrow$ which implies $s \not\leq_c t$ contradicting the assumption, so (3.14) will hold.

$\Longleftarrow$: $\forall \pi \in \Pi(s) : s \leq_\pi t \implies s \leq_\lambda t \implies s \leq_c t.$ $\square$

Corollary 3.87. *Let s, t be Λ-expressions.*

$$(\exists \Psi \subseteq \Pi(s) \forall \rho \in \Psi : s \equiv_\rho t \wedge \forall \pi \in \Pi(s) \setminus \Psi : s <_\pi t) \iff s \leq_c t.$$

In other words the primitive positions of contextually ordered expressions can be partitioned into those being $\equiv_c$-equivalent and those being different but related.
We cannot use the function $f_{s,\pi}$ as a context to distinguish between an expression not having the primitive position π and one which has the the primitive position, but in that position has a divergent expression. We define a function $g_{s,\pi}$, which will converge when applied to an expression t if $\pi \in \Pi(t)$ and which will diverge otherwise. For obvious reasons this definition is very similar to that of $f_{s,\pi}$.

Definition 3.88. Let s be a Λ-expression.

$$g_{s,\lambda} \overset{\text{def}}{=} \mathtt{K\ 1} \tag{3.16}$$

$$g_{s,\mathtt{c}_{A,k}\cdot\imath\cdot\pi} \overset{\text{def}}{=} \lambda x. g_{s_\imath,\pi}\ (\mathtt{case}\ x\ \overbrace{\mathtt{bot}\ldots\mathtt{bot}}^{k-1}\ (\lambda x_1 \ldots x_m . x_\imath)\ \underbrace{\mathtt{bot}\ldots\mathtt{bot}}_{k-1}), \text{if } s\Downarrow_S \mathtt{c}_{A,k}\ s_1 \ldots s_m. \tag{3.17}$$

Lemma 3.89. *Let $s \in \Lambda$, $\pi \in \Pi(s)$, then*

$$(g_{s,\pi}\ t)\Downarrow \iff \pi \in \Pi(t).$$

Proof. Analogous to the proof of lemma 3.82. $\square$

Definition 3.90 ($\widehat{s}$, primitive head). Let s be a Λ-expression. $\mathtt{c}$ is the *primitive head* of s, $\widehat{s} \overset{\text{def}}{=} \mathtt{c}$, iff $s\Downarrow_S \mathtt{c}\ldots$, otherwise s has no primitive head.

Lemma 3.91. *Let s_i be an ascending chain satisfying $\bigsqcup_i^c s_i \equiv_c s$ and let π be a primitive position of s.*

1. $\bigsqcup_i^c f_{s,\pi}\ s_i \equiv_c f_{s,\pi}\ s \equiv_c s_{|\pi}$

2. *There is an index i_0, such that for all indices $i > i_0 : \pi \in \Pi(s_i)$ and either $\widehat{f_{s,\pi}\ s} \equiv \widehat{f_{s,\pi}\ s_i}$ or $f_{s,\pi}\ s$ and $f_{s,\pi}\ s_i$ have no primitive head.*

Proof.

1. The statement follows from the continuity of contexts and from lemma 3.82
2. Using continuity of contexts we see $\bigsqcup^c_i g_{s,\pi}\ s_i \equiv_c g_{s,\pi}\ s$. Lemma 3.89 yields $(g_{s,\pi}\ s)\Downarrow$, we may conclude that $\exists i'_0 : \forall i > i'_0 : (g_{s,\pi}\ s_i)\Downarrow$. Applying lemma 3.89 in the other direction we get $\forall j > i'_0 : \pi \in \Pi(s_j)$.

   If $f_{s,\pi}\ s \equiv_c \texttt{bot}$, then $\forall j > i'_0 : f_{s,\pi}\ s_j \equiv_c \texttt{bot}$ and neither $f_{s,\pi}\ s$ nor $f_{s,\pi}\ s_j$ have a primitive head.

   If $(f_{s,\pi}\ s)\Downarrow_F$, then due to lemma 3.91 (1) and lemma 2.112 we obtain $\exists i_0 : \forall j > i_0 : (f_{s,\pi}\ s_j)\Downarrow_F$. Thus neither $f_{s,\pi}\ s$ nor $f_{s,\pi}\ s_j$ have a primitive head.

   Analogously for $f_{s,\pi}\ s \xrightarrow{*}_{\text{no}} \mathbb{R}[x]$.

   If $(f_{s,\pi}\ s)\Downarrow_S \texttt{c}\ldots$ then due to lemma 3.91 (1) and lemma 2.112 we obtain $\exists i_0 : \forall j > i'_0 : (f_{s,\pi}\ s_j)\Downarrow_S \texttt{c}\ldots$. Thus $\widehat{f_{s,\pi}\ s} \equiv \widehat{f_{s,\pi}\ s_j}$. □

Informally, the following technical lemma states that between a primitive and a non-primitive expression, $\leq_c$-related, we can fit a double chain approximating the latter. This double chain is strictly above the primitive expression wherever that is possible, i.e. where the primitive and the non-primitive expression differ and $\equiv_c$-equivalent elsewhere.

Lemma 3.92. *Let A be a primitive set (cf. definition 3.75), let $s_{ij} \in A$ be a double chain for t, $\bigsqcup^c_j s_{ij} \equiv_c t_i$, $\bigsqcup^c_i t_i \equiv_c t$, let t be*

non-primitive, let s be a primitive Λ-expression with $s \leq_c t$ and let Ψ be the set of differing positions of s and t, then

- *$s <_c t$,*
- *there is a double chain $s'_{ij} \in A$ for t, $\forall \pi \in \Psi : s <_\pi s'_{ij}$ and*
- *$\forall \pi \in \Pi(s) \setminus \Psi : s \equiv_\pi s'_{ij}$.*

Proof. Since t is non-primitive, we must have $s \not\equiv_c t$. From lemma 3.86 we know $\forall \pi \in \Pi(s) : s \leq_\pi t$.
Since $\forall \pi \in \Pi(s) : \pi$ is a primitive position in t, we apply lemma 3.91 to conceive that all but finitely many of the t_i have the primitive positions $\Pi(s)$. The same argument shows for every one of the remaining t_i that all but finitely many of the s_{ij} have the primitive positions $\Pi(s)$. Let us call these r_{ij}.
The r_{ij} form a double chain, such that for every primitive position $\pi \in \Pi(s)$ and every $i, j : \pi$ is a primitive position in r_{ij}.
For every one of the maximal primitive positions π among the $\Pi(s)$ we iteratively modify the r_{ij} to obtain r'_{ij}, which become the r_{ij} for the next iteration according to the following 4 steps.

1. If $t_{|\pi} \equiv_c \texttt{bot}$, then $r_{ij|\pi} \equiv_c \texttt{bot}$, since $r_{ij} \leq_c t$. Consequently, we use $r'_{ij} = r_{ij}$.
2. If $t_{|\pi} \xrightarrow{*} \mathbb{R}[x]$, then for all $i, j : r_{ij|\pi} \xrightarrow{*} \mathbb{R}[x]$ and we use $r'_{ij} = r_{ij}$.
3. If $t_{|\pi}\Downarrow_F$, we reduce the r_{ij} to g_{ij}, such that g_{ij} has sufficient depth to actually have position π not only as a primitive position. This position of the g_{ij} can then be replaced by $f_{t,\pi}\ t$. The g_{ij} become the r'_{ij}. The r'_{ij} are primitive and $\bigsqcup^c_i \bigsqcup^c_j f_{t,\pi}\ r'_{ij} \equiv_c t_{|\pi}$ and since A is a primitive set $r'_{ij} \in A$.

4. If $t_{|\pi}\Downarrow_S \mathtt{c}\ldots$, then either $s\Downarrow_S \mathtt{c}\ldots$ and $\alpha(\mathtt{c}) = 0$, $s_{|\pi} \equiv_c$ $\mathtt{bot}$ or $s_{|\pi} \equiv_c \lambda x.\mathtt{bot}$. In all of these cases we can apply lemma 3.91 twice to choose a double chain r'_{ij} for which $r'_{ij|\pi}\Downarrow_S \mathtt{c}\ldots$.

If π is a differing position of s and t then there is also a differing position ρ of s and t having π as a prefix and which is a maximal primitive position of s. For ρ we cannot have chosen case 1 since $s \leq_\rho t$ must hold and if $t_{|\rho} \equiv_c \mathtt{bot}$ then $s_{|\rho} \equiv_c \mathtt{bot}$ and ρ is not a differing position. If case 2 was chosen then $s_{|\rho} \equiv_c \mathtt{bot}$. Otherwise ρ would not be a differing position and we have $s <_\rho r'_{ij}$. The same holds if for ρ case 3 was chosen. If case 4 was chosen we must have $s_{|\rho} \equiv_c \mathtt{bot}$ or $s_{|\rho} \equiv_c \lambda x.\mathtt{bot}$ and hence $s <_\rho r'_{ij}$. Accordingly, one sees that for primitive positions π which are not differing positions $s \equiv_\pi r'_{ij}$.
The selection of the candidates among the s_{ij} removes finitely many of the t_i and for each of the remaining t_i removes finitely many of the s_{ij}. Lemma 2.121 ensures that these selections do not change the respective club. For every one of the finitely many positions in $\Pi(s)$ we perform the steps 1 – 4. The candidates are not changed by steps 1 and 2. Step 4 selects all but finitely many of the candidates which we already know will not change the clubs. That leaves step 3. Here the candidates are changed, but such that for the inspected primitive position π after the change $r'_{ij} \equiv_\pi t$ holds. Due to lemma 3.86 this entails $\bigsqcup^c_i \bigsqcup^c_j r'_{ij} \equiv_c t$. □

Theorem 3.93. *Let A be a primitive set. Then*

$$\bigsqcup^c \bigsqcup^c A = \bigsqcup^c A.$$

Proof. $\bigsqcup^c A \subseteq \bigsqcup^c \bigsqcup^c A$ is an immediate consequence of the monotonicity of clubs.

We want to show $\bigsqcup^c \bigsqcup^c A \subseteq \bigsqcup^c A$.
The elements $t \in \bigsqcup^c \bigsqcup^c A$ are obtained from double chains $s_{ij} \in A$ with $\bigsqcup^c_j s_{ij} \equiv_c t_i$ and $\bigsqcup^c_i t_i \equiv_c t$. If t or all of the t_i are primitive, membership of t in $\bigsqcup^c_i A$ is evident. We consider the case where neither t nor the t_i are primitive.
Let Π be the differing positions of s_{11} and t. Applying lemma 3.92 we construct a double chain s^1_{ij} for s_{11} satisfying $\forall\pi \in \Pi : s_{11} <_\pi s^1_{ij}$. We form a chain $s_{11} \leq_c s^1_{11} \leq_c s^2_{11}\ldots$ by applying the lemma repeatedly to $s_{11}, s^1_{11}, s^2_{11},\ldots$ and the remaining s^k_{ij}. For this chain if π is a differing position of s^i_{11} and t, then $s^i_{11} <_\pi s^{i+1}_{11}$.
It remains to show that $t \equiv_c \bigsqcup^c_i s^i_{11}$, which would be a consequence of $t \equiv_c \bigsqcup_i s^i_{11}$.
For all $i : s^i_{11} \leq_c t$ and thus $\bigsqcup_i s^i_{11} \leq_c t$.
Assume there is a t' with $t' \equiv_c \bigsqcup_i s^i_{11}$, but $t' <_c t$. From lemma 3.86 we obtain a primitive position $\pi : t' <_\pi t$. Without loss of generality π is a maximal primitive position. Since t has this primitive position, all but finitely many of the s^i_{11} have it as well. This follows from lemma 3.91.
If none of the s^i_{11} satisfy $s^i_{11} \leq_\pi t'$, $\bigsqcup_i s^i_{11} \leq_c t'$ cannot be satisfied either.
If there is an $s^i_{11} \leq_\pi t' <_\pi t$, we distinguish the following cases.

$t'_{|\pi}\Uparrow$: Since $t_{|\pi}\Downarrow$ we see from the construction of the s^i_{11} that $s^{i+1}_{11} \not\leq_\pi t'$, implying $s^{i+1}_{11} \not\leq_c t'$ and hence $\bigsqcup_i s^i_{11} \not\leq_c t'$.

$t'\Downarrow\lambda x.\mathtt{bot}$: If $t_{|\pi}\Downarrow_S$ the argument proceeds as before. Otherwise $t_{|\pi}\Downarrow_F \lambda x.f$ with $\mathtt{bot} <_c f$. Again the construction of the s^i_{11} ensures $s^{i+1}_{11} \not\leq_\pi t'$. □

As an important tool for soundness and completeness proofs we provide the following theorem. It will allow the formulation of the proofs for primitive solutions and to obtain the appropriate statement for the entire concretization. We use the notation $\mathbb{C}[M]$ for $\{\mathbb{C}[m] | m \in M\}$.

Theorem 3.94. *Let D, E be demands.*

$$\mathbb{C}[\eta(D)] \subseteq \gamma(E) \iff \mathbb{C}[\gamma(D)] \subseteq \gamma(E).$$

Proof.

$\Longrightarrow$: Monotonicity of clubs implies $\bigsqcup^c \mathbb{C}[\eta(D)] \subseteq \bigsqcup^c \gamma(E)$. By theorem 3.93 $\gamma(E) = \bigsqcup^c \gamma(E)$. Continuity of contexts (theorem 2.124) implies $\bigsqcup^c \mathbb{C}[\eta(D)] = \mathbb{C}[\bigsqcup^c \eta(D)] = \mathbb{C}[\gamma(D)]$. Summarizing, the statement holds.

$\Longleftarrow$: Since $\eta(D) \subseteq \gamma(D)$. □

We will apply this theorem for soundness proofs and will thus only be required to show that the elements of the representation are indeed solutions and for completeness proofs we will only be required to show that there are no primitive solutions outside the representation of the appropriate demand.

3.6 Demands and Fixpoints

In this section we investigate the relation of concretizations to least and greatest fixpoints of functions. For some demands their definition can be directly translated to a function having a least fixpoint. This is no surprise since $\eta(\cdot)$ is inductively defined and for some demands $D : \eta(D) = \gamma(D)$. For others the same direct translation leads to a function for which the greatest fixpoint

equals the concretization. For yet others the direct translation results in a function with a fixpoint equal to the concretization, but this fixpoint is neither the least nor the greatest.
Superficially, it looks as if the concretization of `Top` is all of Λ. This however is false and we present the following counter-example.

Example 3.95. Let $\mathbb{R}[\cdot]$ be a reduction context and let x be a variable, then $\mathbb{R}[x] \notin \gamma(\mathtt{Top})$. Any expression in $\gamma(\mathtt{Top})$ either converges or diverges, but $\mathbb{R}[x]$ does neither one nor the other.

On the other hand though, there are Λ-expressions with free variables in $\gamma(\mathtt{Top})$, so $\gamma(\mathtt{Top}) \neq \Lambda^0$ as the following example witnesses.

Example 3.96. $\mathtt{K\ 1}\ x \in \gamma(\mathtt{Top})$, since $\mathtt{K\ 1}\ x {\Downarrow_S} \mathtt{1}$.

It is natural to ask which Λ-expressions are indeed in $\gamma(\mathtt{Top})$. $\gamma(\mathtt{Top})$ can be co-inductively characterized. We use notation from [Gor94] in the following theorem and prove that $\gamma(\mathtt{Top})$ is the greatest fixpoint of an obviously monotonous operator on sets.

Theorem 3.97. *Let*

$$F(X) \overset{def}{=} \{t | t{\Uparrow} \vee t{\Downarrow_F} \vee \exists \mathtt{c} \in K : t{\Downarrow_S} \mathtt{c}\ t_1 \ldots t_{\alpha(\mathtt{c})} \wedge \forall i : t_i \in X\}$$

then

$$\nu X.F(X) = \gamma(\mathtt{Top}).$$

Proof. We start by showing that $\gamma(\mathtt{Top})$ is F-dense, i.e. $\gamma(\mathtt{Top}) \subseteq F(\gamma(\mathtt{Top}))$.

dense: Let $s \in \gamma(\mathtt{Top})$, then one of the following cases will occur:

$s{\Uparrow}$: We have $\forall X : s \in F(X)$ and in particular $s \in F(\gamma(\mathtt{Top}))$.

$s\Downarrow_F$: Similar to $s\Uparrow$.

$s\Downarrow_S \texttt{c}\ t_1 \dots t_{\alpha(\texttt{c})}$: With lemma 3.43 we conclude $\forall i : t_i \in \gamma(\texttt{Top})$ and obtain $s \in F(\gamma(\texttt{Top}))$.

least: It remains to show that $\nu X.F(X) \subseteq \gamma(\texttt{Top})$. We proceed as follows: For every $t \in \nu X.F(X)$

1. we construct an ascending chain $s^0 \leq_c s^1 \leq_c \dots$,
2. we show that every chain element is in $\eta(\texttt{Top})$ and
3. we show that $\bigsqcup_i^c s^i \equiv_c t$.

It will then follow that $t \in \gamma(\texttt{Top})$ and the statement is proved.

1. We define

$$s_t^0 \overset{\text{def}}{=} \texttt{bot}$$

$$s_t^i \overset{\text{def}}{=} \begin{cases} \texttt{bot} & \text{if } t\Uparrow, \\ t & \text{if } t\Downarrow_F \text{ and} \\ \texttt{c}\ s_{t_1}^{i-1} \dots s_{t_{\alpha(\texttt{c})}}^{i-1} & \text{if } t\Downarrow_S \texttt{c}\ t_1 \dots t_{\alpha(\texttt{c})}. \end{cases}$$

and use induction to show that the s_t^i form an ascending chain for any t.

$i = 0$: $s_t^0 \equiv \texttt{bot}$ and $\forall r \in \Lambda : \texttt{bot} \leq_c r$.

$i > 0$:

$s_t^i\Uparrow$: $s_t^i \leq_c s_t^{i+1}$ holds.

$s_t^i\Downarrow_F$: $s_t^i \equiv s_t^{i+1}$ holds and implies $s_t^i \leq_c s_t^{i+1}$.

$s_t^i\Downarrow_S \texttt{c}\ s_{t_1}^{i-1} \dots s_{t_{\alpha(\texttt{c})}}^{i-1}$: $t\Downarrow_S \texttt{c}\ t_1 \dots t_{\alpha(\texttt{c})}$ and thus $s_t^{i+1} \equiv \texttt{c}\ s_{t_1}^i \dots s_{t_{\alpha(\texttt{c})}}^i$. From the induction hypothesis we conclude $\forall 1 \leq j \leq \alpha(\texttt{c}) : s_{t_j}^{i-1} \leq_c s_{t_j}^i$ and with lemma 2.95 we get $s_t^i \leq_c s_t^{i+1}$.

2. That every chain element is a member of $\eta(\texttt{Top})$ can similarly be proved by induction.
3. Obviously, $\forall i : s_t^i \leq_c t$. If t is primitive we can show by induction that $t \in \eta(\texttt{Top})$, so in what follows we assume a non-primitive t. In order to apply the criterion of lemma 2.132 it remains to show $\forall \mathbb{C}[\cdot] : \mathbb{C}[t]\Downarrow \implies \exists i : \mathbb{C}[s_t^i]\Downarrow$. For every s_t^i we can reduce t to some t^i, i.e. $t \xrightarrow{*} t^i$, such that s_t^i and t do not differ above argument depth i. Applying theorem 2.59 we conclude $\mathbb{C}[t]\Downarrow \implies \mathbb{C}[t^i]\Downarrow$ and we denote the length of the normal-order reduction of $\mathbb{C}[t]$ to WHNF with n. From $\mathbb{C}[t^i]$ there are at most n normal-order reductions to WHNF, this follows from theorem 2.51. Since t^{n+1} and s_t^{n+1} do not differ above argument depth $n+1$, we can apply the same (at most n) normal-order reductions to $\mathbb{C}[t^{n+1}]$ and to $\mathbb{C}[s_t^{n+1}]$. From $\mathbb{C}[t^{n+1}]$ we reach a WHNF t_W and from $\mathbb{C}[s_t^{n+1}]$ we arrive at s_W. Since t_W and s_W do not differ above argument depth 1, s_W will also be a WHNF. Thus $\bigsqcup_i^c s_t^i \equiv_c t$. □

As a corollary we obtain the

Proposition 3.98. $s \in \Lambda^0 \implies s \in \gamma(\texttt{Top})$.

Proof. $s \in \Lambda^0 \implies s\Uparrow \vee s\Downarrow_F \vee \exists \texttt{c} \in K : s\Downarrow_S \texttt{c}\ s_1 \dots s_{\alpha(\texttt{c})} \wedge \forall i : s_i \in \Lambda^0$. Then for $F(\cdot)$ from theorem 3.97 $s \in F(\Lambda^0)$. Since $\gamma(\texttt{Top})$ is the greatest fixpoint of $F(\cdot)$, $s \in \gamma(\texttt{Top})$. □

The definition of the function $F(\cdot)$ from theorem 3.97 matches the definition of Top very closely. Without presenting a formal defintion of the notion we say that $F(\cdot)$ is a *direct translation* of the demand defintion for `Top`. It will not be this easy to arrive at a co-inductive characterization for a demand in some cases.

Example 3.99. Let $\texttt{Fin} \overset{\text{def}}{=} \langle \texttt{Nil}, \texttt{Top} : \texttt{Fin} \rangle$ and let

$$F(X) \overset{\text{def}}{=} \{t | t \Downarrow_S \texttt{Nil} \vee t \Downarrow_S t_1 : t_2 \wedge t_1 \in \gamma(\texttt{Top}) \wedge t_2 \in X\}$$

then

$$\nu X.F(X) \neq \gamma(\texttt{Fin}).$$

This stems from the fact that there are Λ-expressions in $\nu X.F(X)$ that can be evaluated to arbitrary depth, but these are not in $\gamma(\texttt{Fin})$.
On the other hand with $G(X) \overset{\text{def}}{=} \{t | t \Downarrow_S \texttt{Nil} \vee t \Downarrow_S t_1 : t_2 \wedge t_1 \in \gamma(\texttt{Top}) \wedge t_2 \in X \wedge t_2 \text{ is primitive}\}$ we can satisfy $\nu X.G(X) = \mu X.F(X) = \gamma(\texttt{Fin})$.

The concretization of `Fin` can however be easily characterized as a least fixpoint although its demand definition uses `Top`.

Lemma 3.100. *Let* $F(X) \overset{\text{def}}{=} \{t | t \Downarrow_S \texttt{Nil} \vee t \Downarrow_S t_1 : t_2 \wedge t_1 \in \gamma(\texttt{Top}) \wedge t_2 \in X\}$*, then*

$$\mu X.F(X) = \gamma(\texttt{Fin}).$$

Proof.

$\gamma(\texttt{Fin}) \subseteq \mu X.F(X)$: It is easily shown that $\exists s_i, t_i \in \eta(\texttt{Top}) : s \equiv_c s_1 : s_2 : \ldots : s_n : \texttt{[]} \wedge t \equiv_c t_1 : t_2 : \ldots : t_n : \texttt{[]} \wedge \forall i : s_i \leq_c t_i$ if $s, t \in \eta(\texttt{Fin})$ and $s \leq_c t$. Thus for every $r \in \gamma(\texttt{Fin})$ there is an n and $r_i \in \gamma(\texttt{Top})$ satisfying $r \equiv_c r_1 : r_2 : \ldots : r_n : \texttt{[]}$. That every such r is in $\mu X.F(X)$ can be shown by induction on the number n.

$\gamma(\texttt{Fin}) = F(\gamma(\texttt{Fin}))$: This is easily shown. □

The lemmata and propositions in this section give rise to the hope that we could straightforwardly translate demand definitions into monotonous functions, and to find some property of the demand definitions for the decision whether to use the greatest or the least fixpoint of this function to obtain the demands concretization. Alternatively, but equivalently, we could use this property to decide whether to constrain expression parts to be primitive or not. However, the following observation seems to contradict this possibility.

Example 3.101. Suppose the following demand definitions are given.

$$\texttt{FT} \overset{\text{def}}{=} \langle \texttt{Lf}, \texttt{Br FT IT} \rangle$$
$$\texttt{IT} \overset{\text{def}}{=} \langle \texttt{Bot}, \texttt{Br FT IT} \rangle$$

It seems that due to the mutually recursive definition of the demands a fixpoint for a monotonous function for both of them would need to be formed. The method with which mutually recursive supercombinators are translated to Λ-expressions would be one possibility for forming such a function. We say “seems”, since we do not know of any other methods nor if any could exist in principle. We can, however, observe that the concretization of `FT` and `IT` may be represented as $g(\texttt{FT})$ and $g(\texttt{IT})$ with a fixpoint g of the function

$$F(f) \overset{\text{def}}{=} \lambda x. \begin{cases} \{r | r \Downarrow_S \texttt{Lf} \vee r \Downarrow_S \texttt{Br}\ \ s\ t \wedge s \in f(\texttt{FT}) \wedge t \in f(\texttt{IT})\} \\ \quad \text{if } x \equiv \texttt{FT}, \\ \{r | r \Uparrow \vee r \Downarrow_S \texttt{Br}\ \ s\ t \wedge s \in f(\texttt{FT}) \wedge t \in f(\texttt{IT})\} \\ \quad \text{if } x \equiv \texttt{IT}. \end{cases}$$

Similar to lemma 3.8 we can show that the set $L \overset{\text{def}}{=} \mathcal{P}(\Lambda)^{\{\texttt{FT},\texttt{IT}\}}$ is a complete lattice with the point-wise order. F has fixed points since F is monotonous with respect to this order.
We define

$$g \overset{\text{def}}{=} \lambda x. \begin{cases} \gamma(\texttt{FT}) & \text{if } x \equiv \texttt{FT} \text{ and} \\ \gamma(\texttt{IT}) & \text{if } x \equiv \texttt{IT}. \end{cases}$$

Lemma 3.102. *g is a fixpoint of F.*

Proof.

$\subseteq$: Let $r \in g(\texttt{FT})$ then either $r{\Downarrow_S}\texttt{Lf}$ or $r{\Downarrow_S}\texttt{Br}\ \ s\ t \wedge s \in \gamma(\texttt{FT}) \wedge t \in \gamma(\texttt{IT})$. In the former case obviously $r \in F(g)(\texttt{FT})$, and in the latter case this holds, because $\gamma(\texttt{FT}) = g(\texttt{FT})$ and $\gamma(\texttt{IT}) = g(\texttt{IT})$. For $r \in g(\texttt{IT})$ the proof is almost identical.

$\supseteq$: analogous to $\subseteq$. □

This fixpoint is neither a least nor a greatest fixpoint. Intuitively, the least fixpoint would not include expressions, which can be evaluated infinitely to the right, and the greatest fixpoint would contain expressions that can be evaluated infinitely to the left. The former would lack expressions present in the concretizations and the latter would add expressions not present in the concretizations. Here, constraining $f(\texttt{FT})$ or $f(\texttt{IT})$ to be primitive is insufficient and we would have to come up with entirely new definitions matching the structure of the expressions in the concretizations. We do not go into more detail, but formulate the following conjecture for sensible definitions of *direct translation*.

Conjecture 3.103. *There are demands for which the concretization is neither the least nor the greatest fixpoint of their direct translation to a monotonous function.*

3.7 Demand transformations

Demands produced by our calculi may become quite complicated. In this section we present transformations that can be used to simplify demands while retaining the original demand's concretization. Particularly, these transformations address the simplification of demands defined with `where`-expressions.
This section was motivated by the goal to simplify some demands that actually resulted from demand analyses. For a concrete example the reader is referred to 3.116. Consequently, the transformations are arbitrarily and intentionally chosen and are not complete in any way.
Our first transformation will be applicable in case there is more than one component not directly referring to the name of the result present in the union. We can observe that any of a Λ-expression's approximations in the representation may use an arbitrary number of iterations of the `where`-expressions, but as soon as one of the demand expressions without a `where`-expression is chosen no further iteration is possible. That is to say, that only one of the "base cases" can be used. This observation is expressed more precisely in the following lemma.

Lemma 3.104. *Let $D \overset{def}{=} \langle C_1, \ldots, C_r, D_1\ \texttt{where}\ T_1 = D, \ldots, D_s\ \texttt{where}\ T_s = D\rangle$ where the C_i may contain `where`-expressions, but none that directly reference the name D. Let*

$$E \overset{def}{=} \langle E_1, \ldots, E_r\rangle \text{ where}$$
$$E_i \overset{def}{=} \langle C_i, D_1\ \texttt{where}\ T_1 = E_i, \ldots, D_s\ \texttt{where}\ T_s = E_i\rangle$$

then

$$D \equiv_\gamma E.$$

Proof. We will show $\eta(D) = \eta(E)$. Applying the club to both sides we obtain $\gamma(D) = \gamma(E)$. The two inclusions will be shown separately.

$\subseteq$: Let $t \in \eta(D)$. There is an i for which $t \in \eta_{\Delta^i(\mathbf{0})}(D)$. We prove $\forall t : t \in \eta(D) \implies \exists j : t \in \eta(E_j)$ by contradiction. This then together with lemma 3.38 implies the inclusion. Assume this does not hold, then

$$\exists t, i : t \in \eta_{\Delta^i(\mathbf{0})}(D) \wedge \forall j : t \notin \eta(E_j).$$

Among the expressions for which this holds we choose one, say u, with minimal number i of iterations of Δ. Then

$$\forall k < i : v \in \eta_{\Delta^k(\mathbf{0})}(D) \implies \exists j, m : v \in \eta_{\Delta^m(\mathbf{0})}(E_j). \quad (3.18)$$

$$\begin{aligned}
& u \in \eta_{\Delta^i(\mathbf{0})}(D) \\
\implies & \text{(definition 3.18)} \\
& u \in \eta_{\Delta^{i-1}(\mathbf{0})}(\langle C_1, \ldots, C_r, D_1 \;\texttt{where}\; T_1 = D, \\
& \qquad\qquad\qquad\qquad \ldots, D_s \;\texttt{where}\; T_s = D\rangle) \\
\implies & \text{(lemma 3.38)} \\
& \exists k : u \in \eta_{\Delta^{i-1}(\mathbf{0})}(C_k) \\
& \vee \exists l, \rho : u \in \eta_{\Delta^{i-1}(\mathbf{0})}(\rho D_l) \wedge \rho T_l \sqsubseteq_p^{i-1} D \\
\implies & \text{(lemma 3.26 + (3.18)} \implies \exists j, m : \rho T_l \sqsubseteq_p^m E_j) \\
& \exists k : u \in \eta_{\Delta^{i-1}(\mathbf{0})}(C_k) \\
& \vee \exists l, \rho, j, m : u \in \eta_{\Delta^{i-1}(\mathbf{0})}(\rho D_l) \wedge \rho T_l \sqsubseteq_p^m E_j \\
\implies & \text{(Monotonicity of } \Delta \\
& + \text{ without loss of generality } m \geq i.)
\end{aligned}$$

$$\begin{aligned}
& \exists k : u \in \eta_{\Delta^m(\mathbf{0})}(C_k) \\
& \vee \exists l, \rho, j : u \in \eta_{\Delta^m(\mathbf{0})}(\rho D_l) \wedge \rho T_l \sqsubseteq_p^m E_j \\
\implies & \text{(Lemma 3.38)} \\
& \exists k : u \in \eta_{\Delta^{m+1}(\mathbf{0})}(E_k) \\
& \vee \exists j : u \in \bigcup_l \eta_{\Delta^m(\mathbf{0})}(D_l \;\texttt{where}\; T_l = E_j) \\
\implies & \\
& \exists k : u \in \eta(E_k) \vee \exists j : u \in \eta(E_j)
\end{aligned}$$

in contradiction to the assumption.

$\supseteq$: Due to lemma 3.38 it suffices to show $\forall j : t \in \eta(E_j) \implies t \in \eta(D)$. The rest of the proof proceeds similar to the other inclusion. □

In the following lemmata we find the possibility to narrow the scope of a `where`-expression to the expressions in the contribution, which actually depend on the variables in the pattern. The next lemma starts by stating that a component without variables is not influenced by the `where`.

Lemma 3.105. *Let C be a closed demand and let*

$$B \stackrel{def}{=} [C, E_1, \ldots, E_n] \;\texttt{where}\; [T_1, \ldots, T_m] = D,$$

then $[u_0, \ldots, u_n] \in \eta(B) \iff u_0 \in \eta(C) \wedge [u_1, \ldots, u_n] \in \eta([E_1, \ldots, E_n] \;\texttt{where}\; [T_1, \ldots, T_m] = D)$.

Proof. T abbreviates the pattern $[T_1, \ldots, T_m]$ in the following proof. For some i

$$\begin{aligned}
&[u_0,\dots,u_n] \in \eta_{\Delta^i(\mathbf{0})}([C,E_1,\dots,E_n]\ \texttt{where}\ T = D)\\
\iff&\\
&[u_0,\dots,u_n] \in \bigcup_{\sigma\in\Sigma(T):\sigma T\sqsubseteq_p^i D} \eta_{\Delta^i(\mathbf{0})}(\sigma[C,E_1,\dots,E_n])\\
\iff& (\sigma C \equiv C)\\
&u_0 \in \eta_{\Delta^i(\mathbf{0})}(C)\\
&\wedge [u_1,\dots,u_n] \in \bigcup_{\sigma\in\Sigma(T):\sigma T\sqsubseteq_p^i D} \eta_{\Delta^i(\mathbf{0})}(\sigma[E_1,\dots,E_n])\\
\iff&\\
&u_0 \in \eta_{\Delta^i(\mathbf{0})}(C)\\
&\wedge [u_1,\dots,u_n] \in \eta_{\Delta^i(\mathbf{0})}([E_1,\dots,E_n]\ \texttt{where}\ T = D).
\end{aligned}$$

□

Corollary 3.106. *Let C be a closed demand and let*

$$B \overset{def}{=} [C,E_1,\dots,E_n]\ \texttt{where}\ [T_1,\dots,T_m] = D,$$

then $[u_0,\dots,u_n] \in \gamma(B) \iff u_0 \in \gamma(C) \wedge [u_1,\dots,u_n] \in \gamma([E_1,\dots,E_n]\ \texttt{where}\ [T_1,\dots,T_m] = D)$.

If all expressions from one demand are simply passed through, i.e. the contribution and the pattern are the same and each is a joc having only variables as arguments then we can use the demand itself.

Lemma 3.107. *Let $v_1,\dots,v_n$ be demand variables and let D be a demand, then*

$$\eta_{\Delta^i(\mathbf{0})}([v_1,\dots,v_n]\ \texttt{where}\ [v_1,\dots,v_n] = D) = \eta_{\Delta^i(\mathbf{0})}(D).$$

Proof.

$$\begin{aligned}
&t \in \eta_{\Delta^i(\mathbf{0})}([v_1,\dots,v_n]\ \texttt{where}\ [v_1,\dots,v_n] = D)\\
\iff&\\
&t \in \bigcup_{\sigma\in\Sigma([v_1,\dots,v_n]):\sigma[v_1,\dots,v_n]\sqsubseteq_p^i D} \eta_{\Delta^i(\mathbf{0})}(\sigma[v_1,\dots,v_n])\\
\iff& (\text{Lemma 3.25 + lemma 3.38})\\
&t \in \Delta^i(\mathbf{0})(D) = \eta_{\Delta^i(\mathbf{0})}(D). \quad \square
\end{aligned}$$

In the following lemma we split a `where`-expression into two `where`-expressions: one merely passes expressions to their positions in the original `where`-expression, the other builds all the demands that use variables from the pattern.

Lemma 3.108. *Let*

$$B \overset{def}{=} \langle C, [A,E_2,\dots,E_n]\ \texttt{where}\ [T_1,\dots,T_m] = D\rangle$$

where A is a closed demand and let $v_2,\dots,v_n$ be demand variables and

$$\begin{aligned}
B' &\overset{def}{=} \langle C, [A,v_2,\dots,v_n]\ \texttt{where}\ [v_2,\dots,v_n] = D'\rangle\\
D' &\overset{def}{=} [E_2,\dots,E_n]\ \texttt{where}\ [T_1,\dots,T_m] = D,
\end{aligned}$$

then

$$B \equiv_\gamma B'.$$

Proof. We prove both inclusions separately and it suffices to prove them for η.

$\subseteq$: T abbreviates the pattern $[T_1,\ldots,T_m]$, v abbreviates $[v_1,\ldots,v_n]$ and likewise $E_{2,n}$, $u_{2,n}$ and $v_{2,n}$ abbreviate $[E_2,\ldots,E_n]$, $[u_2,\ldots,u_n]$ and $[v_2,\ldots,v_n]$, respectively.

$$\begin{aligned}
& u \in \eta_{\Delta^i(\mathbf{0})}(B) \\
\Longrightarrow & \\
& u \in \eta_{\Delta^i(\mathbf{0})}(\langle C, [A, E_2, \ldots, E_n] \ \texttt{where}\ T = D\rangle) \\
\Longrightarrow & \\
& u \in \eta_{\Delta^i(\mathbf{0})}(C) \\
& \vee u \in \bigcup_{\sigma\in\Sigma(T):\sigma T \sqsubseteq_p^{i-1} D} \eta_{\Delta^{i-1}(\mathbf{0})}(\sigma[A, E_2, \ldots, E_n]) \\
\Longrightarrow & \text{(Lemma 3.105)} \\
& u \in \eta_{\Delta^i(\mathbf{0})}(C) \vee (u = [u_1,\ldots,u_n] \wedge u_1 \in \eta_{\Delta^i(\mathbf{0})}(A) \\
& \quad \wedge u_{2,n} \in \bigcup_{\sigma\in\Sigma(T):\sigma T \sqsubseteq_p^{i-1} D} \eta_{\Delta^{i-1}(\mathbf{0})}(\sigma E_{2,n})) \\
\Longrightarrow & \text{(Premise)} \qquad (3.19) \\
& u \in \eta_{\Delta^i(\mathbf{0})}(C) \vee (u = [u_1,\ldots,u_n] \wedge u_1 \in \eta_{\Delta^i(\mathbf{0})}(A) \\
& \quad \wedge u_{2,n} \in \eta_{\Delta^i(\mathbf{0})}(D') \\
\Longrightarrow & \text{(Monotonicity of } \Delta \text{ and lemma 3.105)} \\
& u \in \eta_{\Delta^i(\mathbf{0})}(C) \qquad (3.20) \\
& \vee u \in \eta_{\Delta^i(\mathbf{0})}([A, v_2, \ldots, v_n] \ \texttt{where}\ v_{2,n} = D') \\
\Longrightarrow & \\
& u \in \eta_{\Delta^{i+1}(\mathbf{0})}(B').
\end{aligned}$$

$\supseteq$: The proof is analogous to the proof of the inclusion above. □

In some cases a mutual recursion among two demands can be transformed into a recursion over only one demand.

Lemma 3.109. *Let $v_2,\ldots,v_n$ be demand variables and*

$$\begin{aligned}
B &\overset{def}{=} [A, v_2, \ldots, v_n] \ \texttt{where}\ [v_2,\ldots,v_n] = D \\
D &\overset{def}{=} \langle C, [E_2,\ldots,E_n] \ \texttt{where}\ [T_1,\ldots,T_n] = B\rangle
\end{aligned}$$

where $\exists\sigma : \sigma T_1 \sqsubseteq_p A$ and the E_i do not use variables from T_1, then for

$$\begin{aligned}
B' &\overset{def}{=} [A, v_2, \ldots, v_n] \ \texttt{where}\ [v_2,\ldots,v_n] = D' \\
D' &\overset{def}{=} \langle C, [E_2,\ldots,E_n] \ \texttt{where}\ [T_2,\ldots,T_n] = D'\rangle
\end{aligned}$$

$$B \equiv_\gamma B'.$$

Proof. We prove both inclusions separately and it suffices to show $\eta(D) = \eta(D')$.

$\subseteq$: If this would not hold we could choose a u with a least i such that

$$\begin{aligned}
& u \in \eta_{\Delta^i(\mathbf{0})}(D) \wedge u \notin \eta(D') \\
& \quad \wedge (\forall k < i : v \in \eta_{\Delta^k(\mathbf{0})}(D) \implies v \in \eta(D')).
\end{aligned}$$

Again, T abbreviates $[T_1,\ldots,T_n]$, furthermore, $T_{2,n}$, $E_{2,n}$ and $v_{2,n}$ abbreviate $[T_2,\ldots,T_n]$, $[E_2,\ldots,E_n]$ and $[v_2,\ldots,v_n]$, respectively.

$$\begin{aligned}
& u \in \eta_{\Delta^{i-1}(\mathbf{0})}(\langle C, E_{2,n} \ \texttt{where}\ T = B\rangle) \\
\Longrightarrow & \text{(Definition 3.18)}
\end{aligned}$$

$$
\begin{aligned}
& u \in \eta_{\Delta^{i-1}(\mathbf{0})}(C) \cup \bigcup_{\sigma \in \Sigma(T) : \sigma T \sqsubseteq_p^{i-1} B} \eta_{\Delta^{i-1}(\mathbf{0})}(\sigma E_{2,n}) \\
\Longrightarrow\ & \text{(Variables occur only once in the } T_i, \\
& \exists \sigma : \sigma T_1 \sqsubseteq_p A \text{ and no variable from } T_1 \text{ appears} \\
& \text{in the } E_i + \text{lemma 3.105)} \\
& u \in \eta_{\Delta^{i-1}(\mathbf{0})}(C) \\
& \quad \cup \bigcup_{\substack{\sigma \in \Sigma(T_{2,n}), \rho \in \Sigma(T_1) : \Lambda_p(\sigma T_{2,n}) \\ \subseteq \eta_{\Delta^{i-2}(\mathbf{0})}(v_{2,n} \texttt{where} v_{2,n} = D) \\ \wedge \rho T_1 \sqsubseteq_p^{i-2} A}} \eta_{\Delta^{i-1}(\mathbf{0})}(\sigma E_{2,n}) \\
\Longrightarrow\ & \text{(Lemma 3.107, premise)} \\
& u \in \eta_{\Delta^{i-1}(\mathbf{0})}(C) \\
& \quad \cup \bigcup_{\sigma \in \Sigma(T_{2,n}) : \sigma T_{2,n} \sqsubseteq_p^{i-2} D} \eta_{\Delta^{i-1}(\mathbf{0})}(\sigma E_{2,n}) \\
\Longrightarrow\ & \text{(Hypothesis, monotonicity of } \Delta) \\
& \exists j : u \in \eta_{\Delta^{i-1}(\mathbf{0})}(C) \\
& \qquad \cup\, \eta_{\Delta^{j}(\mathbf{0})}(E_{2,n} \ \texttt{where} \ T_{2,n} = D') \\
\Longrightarrow\ & \\
& \exists j : u \in \eta_{\Delta^{j}(\mathbf{0})}(B').
\end{aligned}
$$

$\supseteq$: This inclusion is proved analogously to the one above. □

Remark 3.110. If in lemma 3.109 the condition $\exists \sigma : \sigma T_1 \sqsubseteq_p A$ ensures that expressions from the `where`-expression in D need to be considered at all. If this were not the case that `where`-expression could be entirely ignored.

Lemma 3.111. *Let $v_2, \ldots, v_n$ be demand variables and let A be a*

demand. Furthermore let

$$
\begin{aligned}
B &\overset{def}{=} \langle C, [A, E_2, \ldots, E_n] \ \texttt{where} \ [T_1, \ldots, T_n] = B \rangle \\
B' &\overset{def}{=} \langle C, [A, v_2, \ldots, v_n] \ \texttt{where} \ [v_1, \ldots, v_n] = D' \rangle \\
D' &\overset{def}{=} [E_2, \ldots, E_n] \ \texttt{where} \ [T_1, \ldots, T_m] = B',
\end{aligned}
$$

then

$$B \equiv_\gamma B'.$$

Proof. A proof similar to lemma 3.108 is not sufficient here, but it will suffice to use induction on the number of iterations of Δ. The base case is easy to see. For the induction step we need only change step (3.19) in the proof of lemma 3.108 to employ the induction hypothesis. □

We may shift the "base case", B, from a union consisting of a `where`-expression, W, and B to the demand referenced by W if one component of the contribution is closed and B has the same component. Again, we require some very specific forms of demands.

Lemma 3.112. *Let $v_2, \ldots, v_n$ be demand variables and let the C_i be closed demands. For*

$$
\begin{aligned}
B &\overset{def}{=} \langle [C_1, \ldots, C_n], [C_1, v_2, \ldots, v_n] \ \texttt{where} \ [v_2, \ldots, v_n] = D \rangle \\
D &\overset{def}{=} [E_2, \ldots, E_n] \ \texttt{where} \ [T_1, \ldots, T_m] = E
\end{aligned}
$$

and

$$
\begin{aligned}
B' &\overset{def}{=} [C_1, v_2, \ldots, v_n] \ \texttt{where} \ [v_2, \ldots, v_n] = D' \\
D' &\overset{def}{=} \langle [C_2, \ldots, C_n], [E_2, \ldots, E_n] \ \texttt{where} \ [T_1, \ldots, T_m] = E \rangle
\end{aligned}
$$

we have

$$B \equiv_\gamma B'.$$

Proof. Analogous to lemma 3.108. □

In the same way as the proof of lemma 3.111 refers to the proof of lemma 3.108 the proof of lemma 3.113 refers to the proof of lemma 3.112.

Lemma 3.113. *Let $v_2, \ldots, v_n$ be demand variables, C_i closed demands and*

$$B \overset{def}{=} \langle [C_1, \ldots, C_n], [C_1, v_2, \ldots, v_n] \texttt{ where } [v_2, \ldots, v_n] = D\rangle$$
$$D \overset{def}{=} [E_2, \ldots, E_n] \texttt{ where } [T_1, \ldots, T_m] = B$$
$$B' \overset{def}{=} [C_1, v_2, \ldots, v_n] \texttt{ where } [v_2, \ldots, v_n] = D$$
$$D' \overset{def}{=} \langle [C_2, \ldots, C_n], [E_2, \ldots, E_n] \texttt{ where } [T_1, \ldots, T_m] = B'\rangle,$$

then

$$B \equiv_\gamma B'.$$

Lemma 3.114. *T abbreviates $[T_1, \ldots, T_n]$. Let $C \overset{def}{=} \langle [D_1, \ldots, D_n], [E_1, \ldots, E_n] \texttt{ where } T = C\rangle$ and let $E_1 \equiv T_1$ be a demand variable. Let*

$$C' \overset{def}{=} \langle [D_1, \ldots, D_n], [D_1, E_2, \ldots, E_n] \texttt{ where } T = C'\rangle$$

then

$$C \equiv_\gamma C'.$$

Proof. In the same way as for the other induction proofs for demand transformations. □

Now we have set the stage for proving the

Proposition 3.115. *Let $v_2, \ldots, v_n$ be demand variables,*

$$C \overset{def}{=} \langle [D_1, \ldots, D_n], [E_1, \ldots, E_n] \texttt{ where } [T_1, \ldots, T_n] = C\rangle$$

where $E_1 \equiv T_1$ is a demand variable not present in the remaining E_i

$$C' \overset{def}{=} [D_1, v_2, \ldots, v_n] \texttt{ where } [v_2, \ldots, v_n] = D'$$
$$D' \overset{def}{=} \langle [D_2, \ldots, D_n], [E_2, \ldots, E_n] \texttt{ where } [T_2, \ldots, T_n] = D'\rangle$$

then

$$C \equiv_\gamma C'.$$

Proof. From lemma 3.114 we obtain $C \equiv_\gamma C_1$ where $C_1 \overset{def}{=} \langle [D_1, \ldots, D_n], [D_1, E_2, \ldots, E_n] \texttt{ where } [T_1, \ldots, T_n] = C\rangle$.
With lemma 3.111 it follows that $C_1 \equiv_\gamma C_2$ where $C_2 \overset{def}{=} \langle [D_1, \ldots, D_n], [D_1, v_2, \ldots, v_n] \texttt{ where } [v_2, \ldots, v_n] = C_2'\rangle$ and $C_2' \overset{def}{=} [E_2, \ldots, E_n] \texttt{ where } [T_1, \ldots, T_n] = C_2$.
With lemma 3.113 we get $C_2 \equiv_\gamma C_3$ where $C_3 \overset{def}{=} [D_1, v_2, \ldots, v_n] \texttt{ where } [v_2, \ldots, v_n] = C_3'$ and $C_3' \overset{def}{=} \langle [D_2, \ldots, D_n], [E_2, \ldots, E_n] \texttt{ where } [T_1, \ldots, T_n] = C_3\rangle$.
Finally, by lemma 3.109 we have $C_3 \equiv_\gamma C_4$ where $C_4 \overset{def}{=} [D_1, v_2, \ldots, v_n] \texttt{ where } [v_2, \ldots, v_n] = C_4'$ and $C_4' \overset{def}{=} \langle [D_2, \ldots, D_n], [E_2, \ldots, E_n] \texttt{ where } [T_1, \ldots, T_n] = C_4'\rangle$. □

Example 3.116. As an example for applying demand transformations we present the result of analyzing $\texttt{append}\ xs\ ys \in \texttt{Inf}$, i.e.

$$D \overset{\text{def}}{=} \langle [\texttt{Bot}, \texttt{Top}], [\texttt{[]}, \texttt{Inf}], [\texttt{Top} : zs, ys]\ \texttt{where}\ [zs, ys] = D\rangle.$$

This result will now be simplified. Lemma 3.104 yields $E \equiv_\gamma D$ for

$$E \overset{\text{def}}{=} \langle E_1, E_2\rangle$$

$$E_1 \overset{\text{def}}{=} \langle [\texttt{Bot}, \texttt{Top}], [\texttt{Top} : zs, ys]\ \texttt{where}\ [zs, ys] = E_1\rangle$$

$$E_2 \overset{\text{def}}{=} \langle [\texttt{[]}, \texttt{Inf}], [\texttt{Top} : zs, ys]\ \texttt{where}\ [zs, ys] = E_2\rangle.$$

We apply proposition 3.115 to E_1 and E_2 and obtain $E' \equiv_\gamma E$ with

$$E' \overset{\text{def}}{=} \langle E_1', E_2'\rangle$$

$$E_1' \overset{\text{def}}{=} [v, \texttt{Top}]\ \texttt{where}\ v = F_1$$

$$F_1 \overset{\text{def}}{=} \langle \texttt{Bot}, \texttt{Top} : zs\ \ \texttt{where}\ \ zs = F_1\rangle$$

$$E_2' \overset{\text{def}}{=} [v, \texttt{Inf}]\ \texttt{where}\ v = F_2$$

$$F_2 \overset{\text{def}}{=} \langle \texttt{[]}, \texttt{Top} : zs\ \texttt{where}\ zs = F_2\rangle.$$

Remember that $F_1 \equiv_\gamma \texttt{Inf}$ and $F_2 \equiv_\gamma \texttt{Fin}$. With lemma 3.105 and lemma 3.107 we can obtain $E'' \equiv_\gamma E'$ where

$$E'' \overset{\text{def}}{=} \langle E_1'', E_2''\rangle$$

$$E_1'' \overset{\text{def}}{=} [\texttt{Inf}, \texttt{Top}]$$

$$E_2'' \overset{\text{def}}{=} [\texttt{Fin}, \texttt{Inf}].$$

The result of analyzing $\texttt{append}\ xs\ ys \in \texttt{Inf}$ is thus $\equiv_\gamma$-equivalent to $\langle [\texttt{Inf}, \texttt{Top}], [\texttt{Fin}, \texttt{Inf}]\rangle$.

4 Demand-Analysis

In this chapter we define demand-analysis and present two tableau-based calculi for its partial computation.
The goal of demand-analysis is to find substitutions σ for a given Λ-expression s and a given demand D that send s to an expression in D, i.e. for which $\sigma s \in \gamma(D)$. The analysis' result is a demand containing these substitutions.
Demand-analysis is a backwards analysis using demands to represent sets of Λ-expressions and determining which Λ-expressions are admissible as sub-expressions of an entire Λ-expression in order to make it a member of a given demand [Hug88, HL92, Wra85]. In contrast to these works we do not use an abstract domain, but sets of Λ-expressions.
Demand-analysis is also an inverse computation [GS96, Sør96] since for the Λ-expressions s in the demand for the entire expression, bindings for the inputs are computed such that the resulting expression is equivalent to s.

4.1 Tableaux

In this chapter we present two calculi for demand analysis: one (ADE) is sound and in general incomplete and one (CADE) is sound and complete (if it terminates). Each is a tableau calculus and as such uses a tree as its data structure and applies expansion rules to this tree. Each expansion rule can only be

applied under specific conditions. If these hold, its application attaches new leaves to a leaf of the tree. This process continues non-deterministically until either all the leaves are in a sufficiently simple form, i.e. are *closed*, or the resources are exhausted.
An important concept in this dissertation is that of *tableau calculi*. A tableau calculus can be distinguished by applications of expansion rules, which consider only one leaf to attach new leaves. Successive application of expansion rules may produce an infinite tableau. In a subsequent phase rules are applied which take into account the entire path from the root to a node. These rules can be used to find loops, i.e. repeating sub-tableaux and thereby allow us to work with only a finite portion of an infinite tableau. We will use tableau calculi to present the demand analysis (cf. 4.1.1) itself, which is at the heart of this work, but they are also used e.g. in the implementation to approximate the property `bot`-closed. For these similar problems they seem a natural choice.

4.1.1 Demand tableaux

The data structure we choose for stating our calculi are demand-tableaux. These consist of three components: one representing the information to conveniently apply rules, one compressing the information into a single demand-expression for programmer feedback or optimizer access and one collecting the free variables in the input. Only the information in the first component is essential, the information in the other two could be obtained by projections of the first.

Definition 4.1 (demand-tableaux). A *demand-tableau* $\mathcal{T}$ consists of a *tree representation* (or simply *tree*) $\mathcal{T}_T$, a *standard representation* $\mathcal{T}_N$ and the *root variables* $\mathcal{T}_\mathcal{V} = (x_1, \ldots, x_n)$. $\mathcal{T}_T$ is a labeled tree, in which nodes are labeled with sets of *constraints*. The constraints have the form $s \in D$, where $s \in \Lambda$ and $D \in \Lambda_C$. A constraint with a left hand side consisting of a single variable is called *variable constraint* (or *VC* for short). Additionally, a node may have at most one of the labels "*no!*" or "*loop!* S `where` $T = N$". The edges of $\mathcal{T}_T$ are labeled with substitutions having substitutes formed with constructors, variables and `bot`. These substitutions may either be *id* or else the variables in their substitutes must be fresh, i.e. they must not occur anywhere else in $\mathcal{T}_T$. The *substitution along a path* $\mathcal{P}$ in $\mathcal{T}_T$ is then $\{x_i \mapsto \pi_m \dots \pi_1 x_i | i \in \{1, \dots, n\}\}$ where the π_i are the edge labels along $\mathcal{P}$ with π_1 being closest to the root. $\mathcal{T}_N$ is a demand definition assigning a unique name N to the analysis result. In general the demand thus assigned will be the union of demand expressions that may reference N.

Remark 4.2. Since the variables in the (non-trivial) substitutes of the edge labels must be fresh, the substitution along a path in a tableau will be idempotent.

Notation 4.3. With $\mathcal{T}_\mathcal{V} = (x_1, \dots, x_n)$ we can conceive of every n-tuple $(s_1, \dots, s_n)$ as a substitution $\{x_1 \mapsto s_1, \dots, x_n \mapsto s_n\}$. We write $(s_1, \dots, s_n)_{(x_1,\dots,x_n)}$ or $(s_1, \dots, s_n)_{\mathcal{T}_\mathcal{V}}$ in this case. Conversely, we can conceive of every substitution $\sigma = \{x_1 \mapsto s_1, \dots, x_n \mapsto s_n\}$ as an n-tuple $(s_1, \dots, s_n) = \sigma\mathcal{T}_\mathcal{V}$.

Notation 4.4. Simplifying our presentation we will frequently identify the nodes with their labels. Furthermore we write $(s \in D) \in N$ for a node N and a constraint $s \in D$ to indicate that $\{s \in D, \dots\}$ is the label of node N. Edges labeled σ from a node M to a node N are treated similarly and we say that N is *reachable* from M via σ. We drop edge labels if they are *id*.

Notation 4.5. Demands have no free variables, hence we use $\mathcal{FV}(N)$ for a node N to mean $\bigcup_{(s \in D) \in N} \mathcal{FV}(s)$.

Standard representation

Expansion rules are applied to the tree-representation, but the standard representation is much more concise and is thus preferred if defined. The standard representation is only defined for distinguished tableaux, namely *closed* tableaux and only represents the result of applying one of our calculi, but no intermediate steps. Intuitively, closed tableaux are sufficiently simplified for a result to be concisely stated. In closed tableaux no further "simplification" is necessary, either because the leaf does not have any solution at all and this is detected rendering all attempts for further simplification unnecessary, or because rules in the sub-tableau could be repeated over and over below the leaf, or because the constraints are solved in the sense that only single variables appear in their left hand sides. In all of these cases we can concisely represent the substitutions solving the root constraints as a single demand. Formally, we define a closed tableau below.

Definition 4.6 (closed tableau). A tableau is called *closed*, if all leaves in its tree are either

1. labeled *no!*, or
2. labeled *loop!* S `where` $T = N$, or
3. consist of 0 or more VCs only.

For a closed tableau the standard representation consists of contributions from 2. and 3. and the following definition specifies how these leaves contribute to the standard representation.

Definition 4.7 (contribution to the standard representation). Leaves from 2. and 3. in definition 4.6 *contribute* one component each *to the standard representation.* Recall that the standard representation, $\mathcal{T}_N$, is a demand definition for a name N. Its right hand side consists of a union to which leaves from 2. contribute precisely S `where` $T = N$. Leaves, R, from 3. contribute $\top\sigma_R\sigma\mathcal{T}_\mathcal{V}$, where $\top$ is the substitution mapping every variable to `Top`, σ_R is the substitution which substitutes the demand C for every VC $x \in C$ in R and σ is the substitution along the path from the root to the leaf.

It is important to note, that the standard representation is defined based on the set of open variables $\mathcal{T}_\mathcal{V}$ at the root, i.e. that root variables occurring free more than once will use only one component of the standard representation.

Root variables

As mentioned before the goal of ADE and CADE is to find closed substitutions substituting values for the free variables of the input expression which make the resulting expressions members of their appropriate concretization, so obviously identical variables will receive the same value. Some of our expansion rules will make assumptions about the structure of the free variables in the constrained Λ-expansion, i.e. they will partially specify the value of the variable. These assumptions are recorded in the edge labels and are applied to all the constraints in the descendant with fresh variables in place of any possible sub-structure. Consequently, the goal changes from finding substitutions sending the root expressions to their appropriate concretization to finding such substitutions for the free variables in the substitutes of the substitution

along the path, i.e. the *constrained variables.*

Definition 4.8 (constrained variables). Let $\mathcal{T}$ be a tableau, let N be a node in $\mathcal{T}_T$ and let σ be the substitution along the path from the root to N. The *constrained variables of the node* N, $\mathcal{CV}(N)$, are the free variables of images of the root variables under σ, i.e. $\mathcal{CV}(N) \overset{\text{def}}{=} \mathcal{FV}(\sigma\mathcal{T}_\mathcal{V})$.

Constrained variables of a node N may be absent from all the constraints of N.

Example 4.9. Let $\mathcal{T}$ be a tableau for the input `K` x $y \in$ `Bot`, in which the root has the successor N labeled $x \in$ `Bot`. The variable y is constrained at node N since $\{x, y\} = \mathcal{FV}(\sigma\mathcal{T}_\mathcal{V})$, but it does not appear there anymore.

Graphs of tableaux tend to quickly become large, so while we will try to be explicit about every expansion rule applied, if it serves comprehensibility we will compress the tableau by presenting some steps as a *big step.* In this case we will connect two adjacently drawn nodes by a dashed line, possibly annotated with the sequence of rules applied written left to right.
The variables occurring free in expressions essentially differ from those which occur free in a deeper sub-expression, but are bound further up. The root variables, $\mathcal{T}_\mathcal{V}$, stand for positions in the input expression for which there are no assumptions, but at which concrete sub-expressions from Λ^0 are placed just that nothing yet is known about their structure. The same is true for the free variables of expressions in the constraints deeper on the tableau. Alternatively, we could formulate the expressions in the constraints as elements of Λ^0 with a set of associated positions, each of which specifying that nothing is assumed about the structure of the sub-expression below it. One disadvantage of this alternative is that

the implicit equality of sub-expressions can not be captured as obviously as with variables of the same name. Yet another way to view the expressions in the constraints is as multi-contexts in which holes may be used more than once.

Rules

Rules define how to transform tableaux. If a rules premises are met it can be applied to generate a new tree from a given tree by attaching new leaves below a leaf found in the given tree. We distinguish rules according to the scope needed for their premises: those which only consider the constraints of the predecessor are termed *local rules* and those which consider an entire path from the root to some node are termed *loop rules*, since the only such rules we present are for finding potential loops in the tableau. While it would be possible to include other rules having a scope larger than a leaf in the calculi we have not found a motivation to do so from our experiments.

Definition 4.10 (rule). A *rule* consists of its *premises* and its *consequents*. Usually, we write rules as in a Gentzen calculus: the premise above a horizontal line and the consequents below that line separated by "|". It may however be necessary to additionally relate parts of the premise with parts of a consequent. Such relations are written to the right of the sequent if space permits and are otherwise relegated to the surrounding text. Unless otherwise noted the edge label *id* is associated with a rule.
A *rule is applicable* if all its premises are met.
The *application of a rule* generates a new tableau $\mathcal{T}'$ from a given tableau $\mathcal{T}$ by appending new leaves N_i to a leaf N in $\mathcal{T}_T$ with edges marked according to the edge label associated with the rule.

The notion of a *narrow path* provides a collection of all those rules on the path from the root to a node, which contributed in the formation of some constraint $r \in D$ in N.

Definition 4.11 (narrow path). Let M, N be nodes in the tree $\mathcal{T}_T$ of a tableau, let σ be a substitution along the path from M to N and let $r \in D$ be one of the constraints at M. There is a *narrow path* from $r \in D$ in M to $r' \in D'$ in N, if

1. N is the immediate successor of M and
   a) the rule applied does not replace $r \in D$ and $r \equiv r'$ and $D \equiv D'$ or
   b) the rule replaces $r \in D$ by $r_1 \in D_1, \ldots, r_n \in D_n$ and $\exists i : r_i \equiv r'$ and $D_i \equiv D'$ or
2. there is some L and a narrow path from $r \in D$ in M to $r_1 \in D_1$ in L and from $r_1 \in D_1$ in L to $r' \in D'$ in N.

See figure 4.1.

ADE and CADE

Definition 4.12 (ADE, CADE). CADE and ADE are tableau calculi expanding demand tableaux using expansion rules until the tableau is closed (or until resources are exhausted). Both calculi use different sets of rules (cf. sections 4.4 and 4.5):

CADE uses: (red), (decomp), (redtop), (wc), (jocdec), (redbot), (nobot), (redfun), (nofun), (union), (is), (casep), (c-loop), (loopdecomp), (loopred), (noloop)

ADE uses: the same rules as CADE, but instead of (c-loop) ADE uses (loop) and additionally (type) and (reuse).

$$\begin{tikzcd}[column sep=large]
{} \arrow[d, dash, dashed] & \\
\{r \in D \dots\} \arrow[d, dash, dashed, "\text{rule replacing } r{\in}D \text{ with the } r_i{\in}D_i"] & (M) \\
\{r_1 \in D_1, \dots, r_n \in D_n \dots\} \arrow[d, dash, dashed, "\text{narrow path from } r_1{\in}D_1 \text{ to } r'{\in}D'"] & (L) \\
\{\dots, r' \in D', \dots\} & (N)
\end{tikzcd}$$

Figure 4.1: A narrow path from $r \in D$ in M to $r' \in D'$ in N

We will later prove that while ADE and CADE are both sound, CADE is also complete but ADE generally is not.

4.2 Solutions

4.2.1 Deeply well-typed

We assume the high-level language analyzed to be statically and strongly typed. From this assumption we deduce that programs will not have run-time type errors. This assumption is motivated by our goal to make demand-analysis independent of a specific type-system on the one hand while allowing our calculi to exploit the fact that programs will not have run-time type errors. While expressions which are ill-typed are contextually equivalent to `bot`, we allow the omission of ill-typed solutions since we assume a statically typed language and otherwise the analysis result

of e.g. `length` $xs \in$ `Bot` would need to include many solutions which only confuse the user and distract from the interesting solutions. In the example, the solution $\{xs \mapsto$ `True`$\}$ would also need to be included among many others.

Example 4.13. For the analysis of `length` $xs \in$ `Bot` we obtain the result `Inf` $=$ $\langle$`Bot`, `Top : Inf`$\rangle$ with our calculi. If we would not restrict the calculi to satisfy some typing constraint for the results, a complete solution would have to substitute FWHNFs as well as expressions having SCWHNFs with top level constructors not suitable for constructing lists. Obviously, such a result would contribute less to the programmer's understanding of the program at hand than the "type-cleaned" result. What's more, these solutions would lead to expressions that would not have passed type check. To make matters worse, such a result would not be well suited for further automatic analysis and optimization, since the increase in result size increases the branching degree in tableaus using the result, `case`-branches would need a branch for every constructor in the program instead of a branch for every constructor of the appropriate type-constructor.

While for some analyses it suffices to try evaluation of the Λ-expression in a constraint to WHNF, e.g. for $s \in$ `Bot` or $s \in$ `Top : Top`, others may need normal-order reduction of a constructor argument e.g. $s : t \in$ `Top : Top : Top` or $s : t \in$ `Top : []`. For analyses proceeding to constructor arguments we would then have the same problem discussed above: `case`-branches for every constructor in the program would need to be provided otherwise even simple analyses would be incomplete. We decide to define completeness for a subset of the solutions making the root expression *deeply well-typed*.

Definition 4.14 (deeply well-typed).

$$\begin{aligned} WT_0 &\overset{\text{def}}{=} \Lambda \\ WT_n &\overset{\text{def}}{=} \{t | t\Downarrow_S \mathtt{c}\ t_1 \dots t_{\alpha(\mathtt{c})} \wedge t_i \in WT_{n-1}\} \\ &\cup \quad \{t | t\Downarrow_F\} \\ &\cup \quad \{t | t\Uparrow \wedge t \in WT\} \end{aligned}$$

We define $WT_\Downarrow \overset{\text{def}}{=} \bigcap_i WT_i$ and call the expressions in $WT_\Downarrow$ *deeply well-typed*.

There is no problem in defining WT_i and $WT_\Downarrow$ in this way: using techniques we have demonstrated elsewhere (section 3.6 and chapter 3) it is straightforward to define a monotonous operator $W(\cdot)$ on sets of Λ-expressions, and to deduce the existence of a greatest fixpoint with the Theorem of Knaster and Tarski and the iterative characterization with the CPO Fixpoint Theorem I.
We chose the co-inductive definition in order to include non-primitive solutions, e.g. $\{x \mapsto$ `repeat 1`$\}$.
Some properties of $WT_\Downarrow$ will be considered next.

Theorem 4.15 (Invariance of $WT_\Downarrow$). *Let $s, t \in \Lambda$ and let $s \to t$, then*

$$s \in WT_\Downarrow \iff t \in WT_\Downarrow.$$

Proof. We show: for all $s, t \in \Lambda$ with $s \to t$ and all $i : s \in WT_i \iff t \in WT_i$.

$\Longrightarrow$: Assume this does not hold, then there is a smallest j for which s and t exist satisfying $s \to t$ and $s \in WT_j \wedge t \notin WT_j$, but for all smaller $k : s \to t \wedge s \in WT_k \implies t \in WT_k$. We distinguish the following cases for s:

$s\Downarrow_S \mathtt{c}\ s_1 \dots s_{\alpha(\mathtt{c})} \wedge \forall i : s_i \in WT_{j-1}$: We know $t\Downarrow_S \mathtt{c}\ t_1 \dots t_{\alpha(\mathtt{c})}$ and $s_i \xrightarrow{*} t$, this in a consequence of the standardization theorem. Thus $t_i \in WT_{j-1}$ which implies $t \in WT_j$ contradicting the assumption.

$s\not\Downarrow \wedge s \in WT$: By theorems 2.59 and 2.75 both properties are invariant with respect to reduction so $t \in WT$ and $t\not\Downarrow$.

$s\Downarrow_F$: As for $s\not\Downarrow \wedge s \in WT$.

$\Longleftarrow$: The argument is as for the other implication.

Obviously, the above implies $s \in WT_\Downarrow \iff t \in WT_\Downarrow$. □

Remark 4.16. Diverging well-typed Λ-expressions are deeply well-typed and so are Λ-expressions with an FWHNF.

Lemma 4.17. $\mathtt{c}\ s_1 \dots s_{\alpha(\mathtt{c})} \in WT_\Downarrow$, *iff* $\forall i : s_i \in WT_\Downarrow$.

Proof.

$\Longrightarrow$: Assume this does not hold. There must be an i, for which $s_i \notin WT_\Downarrow$, but $\mathtt{c}\ s_1 \dots s_{\alpha(\mathtt{c})} \in WT_\Downarrow$. Then there has to be a j with $s_i \notin WT_j$. By definition of WT_{j+1} we obtain $\mathtt{c}\ s_1 \dots s_{\alpha(\mathtt{c})} \notin WT_{j+1}$. This contradicts the assumption and the statement holds.

$\Longleftarrow$: $WT_\Downarrow$ is a fixpoint and $\mathtt{c}\ s_1 \dots s_{\alpha(\mathtt{c})} \in W(X)$ if all $s_i \in X$, so $\mathtt{c}\ s_1 \dots s_{\alpha(\mathtt{c})} \in WT_\Downarrow$ if $\forall i : s_i \in WT_\Downarrow$. □

Lemma 4.18. $WT_\Downarrow \subseteq WT$.

Proof. Assume this is false, i.e. $\exists t \in WT_\Downarrow : t \in IT$. From the definition of IT (2.67) it follows that there is a $t' \in \Lambda$ for which $t \xrightarrow{*}_{\text{no}} t' \equiv \mathbb{R}[s] \wedge s \in DIT$ holds.

Here $\mathbb{R}[\cdot]$ cannot be the trivial reduction context, since then $s \equiv t$ and because of $t \in WT_{\Downarrow}$ it would be that $s \equiv \mathtt{c}_{A,i}\ t_1 \dots t_{\alpha(\mathtt{c}_{A,i})}$, i.e. $s \notin DIT$.
Since $t' \equiv \mathbb{R}[s]$ due to definition 2.25 t' cannot normal-order reduce any further and since $\mathbb{R}$ is non-trivial t' cannot be of the form $\mathtt{c}_{A,i}\ t_1 \dots t_{\alpha(\mathtt{c}_{A,i})}$, because then $s \equiv \mathtt{c}_{A,i} \wedge \mathbb{R}[\cdot] \equiv [\cdot]\ t_1 \dots t_{\alpha(\mathtt{c}_{A,i})}$ would have to hold, but $\mathtt{c}_{A,i} \notin DIT$. □

A Λ-expression cannot be deeply well-typed if an ill-typed expression can be found at any of its primitive positions (cf. definition 3.68).

Lemma 4.19. *Let* $s \in \Lambda$, *then*

$$\forall \pi \in \Pi(s) : s_{|\pi} \in IT \implies s \notin WT_{\Downarrow}.$$

Proof. The proof is by induction on π.

$\pi = \lambda :$ In this case $s \in IT$. By lemma 2.73 we obtain $s\Uparrow$. It must be that $s \notin WT_{\Downarrow}$, because $s \in WT_{\Downarrow} \wedge s\Uparrow \implies s \in WT$.

$\pi = \mathtt{c} \cdot \imath \cdot \rho :$ Obviously, $s\Downarrow_S \mathtt{c}\ s_1 \dots s_{\alpha(\mathtt{c})}$ and $s_{\imath|\pi} \in IT$. With the induction hypothesis we conclude $s_\imath \notin WT_{\Downarrow}$ and with lemma 4.17 we obtain the statement. □

Lemma 4.20. *If* $\mathbb{D}[s] \in WT_{\Downarrow}$, *then* $\mathbb{D}[\mathtt{bot}] \in WT_{\Downarrow}$.

Proof.

$\mathbb{D}[s]\Uparrow :$ Due to lemma 2.90 $\mathbb{D}[\mathtt{bot}]\Uparrow$. In this case $\mathbb{D}[s] \in WT_{\Downarrow}$ amounts to $\mathbb{D}[s] \in WT$ and we conclude with corollary 2.81 that $\mathbb{D}[\mathtt{bot}] \in WT$ and therefore $\mathbb{D}[\mathtt{bot}] \in WT_{\Downarrow}$.

$\mathbb{D}[s]\Downarrow :$ We use induction on i to show $\mathbb{D}[s] \in WT_i \implies \mathbb{D}[\mathtt{bot}] \in WT_i$.

$i = 0 :$ $WT_0 = \Lambda$, thus the statement trivially holds.

$i > 0 :$ There is a normal-order reduction sequence from $\mathbb{D}[s]$ to WHNF. If $\mathbb{D}[s]$ and $\mathbb{D}[\mathtt{bot}]$ already differ at argument depth 0 then $\mathbb{D}[\cdot]$ is a reduction context and $\mathbb{D}[\mathtt{bot}]\Uparrow$. According to corollary 2.81 $\mathbb{D}[\mathtt{bot}] \in WT$ and thus $\mathbb{D}[\mathtt{bot}] \in WT_i$. If, on the other hand, $\mathbb{D}[s]$ and $\mathbb{D}[\mathtt{bot}]$ are equivalent above some argument depth $n > 0$ their normal-order redex is at the same position and has to be reduced by the same alternative of the $\rightarrow_B$-reduction. We iterate this distinction until either the former situation eventuates or until all the normal-order reductions of $\mathbb{D}[s]$'s sequence have been applied to $\mathbb{D}[\mathtt{bot}]$ and the contracti are still equivalent above some argument depth $m > 0$, whichever comes first. In the latter case the contractum of $\mathbb{D}[\mathtt{bot}]$ is also a WHNF. If both contracti are FWHNFs, the statement is proved. Otherwise, both will be SCWHNFs using the same constructor. Assuming $\mathbb{D}[s] \in WT_i$, we can apply the induction hypothesis for the constructor arguments to conclude the statement. □

4.2.2 Solution

We split the definition of solutions into the definition of solution, which satisfies the constraint at a node including the implicit constraints and the definition of c-solution which additionally makes the root expression deeply well-typed. For completeness the set of c-solutions will be considered.

Definition 4.21 ($U(\cdot)$, solution). A node M in some tableau $\mathcal{T}$ labeled with the additional label *no!* has *no solution* at all. If M

has no additional label and its constraints are $\{s_1 \in C_1, \ldots, s_n \in C_n\}$ then a ground substitution θ is a *solution of the node M*, iff $\forall i : \theta s_i \in \gamma(C_i)$, $\forall x \in \mathcal{CV}(M) \setminus \mathcal{FV}(M) : \theta x \in \gamma(\texttt{Top})$ and $\mathrm{dom}(\theta) = \mathcal{CV}(M)$. If N will be the demand name used by the standard representation and an additional label *loop!* S `where` $T = N$ is present θ will also need to be a member of $\gamma(S\ \texttt{where}\ T = N)_{\mathcal{T_V}}$ to be a solution of M.
The set of solutions of R is written $U(R)$.

Definition 4.22 ($U^c_{\mathcal{T}}(\cdot)$, c-solution). Let $\mathcal{T}$ be a tableau with root constraint $\{s_1 \in D_1, \ldots, s_m \in D_m\}$ and let σ be the (idempotent) substitution along the path to a node, L, having solution θ. θ is a *c-solution of L*, iff $\theta\sigma s_i \in WT_{\Downarrow}$.
The set of c-solutions of L is written $U^c_{\mathcal{T}}(L)$.

For the proofs of soundness and completeness it will be advantageous to define *primitive solutions*, i.e. solutions which can be seen to belong to the concretization without resorting to the use of clubs.

Definition 4.23 ($U_\eta(\cdot)$, primitive solution). Let everything be as in definition 4.22 with the only difference, that

$$\exists D \in \Lambda_C : \theta\mathcal{T_V} \in \eta(D)$$

where $\mathcal{T_V}$ are the root variables of the tableau then θ is called a *primitive solution of L.*
For the set of all such solutions of L we write $U_\eta(L)$.

Primitive solutions are required to substitute primitive terms, but the expressions resulting from substitution do not need to be primitive. In contrast, the latter *is* required for *very primitive solutions.*

Definition 4.24 ($U_\dagger(\cdot)$, very primitive solutions). Let everything be as in definition 4.21 with the only difference, that

$$\forall i : \theta s_i \in \eta(C_i)$$

then θ is called a *very primitive solution of the node.*
For the set of all such solutions of a node we write $U_\dagger(R)$.

Lemma 4.25. *$\theta \in U(R) \implies U_\eta(R) \neq \emptyset$.*

Proof. Let R be a node labeled $\{s_1 \in D_1, \ldots, s_n \in D_n\}$. $\theta \in U(R) \implies \theta s_i \in \gamma(D_i)$. If θs_i is primitive, then $\theta s_i \in \eta(D_i)$ and there is a θ' with $\theta' s_i \equiv_c \theta s_i$ and $\theta' \in U_\eta(R)$. If $\theta\vec{x}$ is non-primitive, where $\vec{x}$ are the free variables in R, then there is an ascending chain of expressions $\theta_1\vec{x} \leq_c \theta_2\vec{x} \leq_c \ldots$ for which $\theta\vec{x} \equiv_c \bigsqcup_i^c \theta_i\vec{x}$. The θ_i are primitive with $\theta_i s_j \in \eta(D_j)$. Thus $\theta_i \in U_\eta(R)$. □

Lemma 4.26. *There are sets of constraints R, such that $U(R) \neq \emptyset \wedge U_\dagger(R) = \emptyset$.*

Proof. An example is the constraint $R \overset{\text{def}}{=} \texttt{repeat}\ x \in \texttt{Inf}$. For $\theta = \{x \mapsto \texttt{1}\}$ we obtain $\theta \in U(R)$ but $\forall\theta : \theta(\texttt{repeat}\ x) \notin \eta(\texttt{Inf})$. □

The definition of the semantics of the standard representation requires that its members make the root deeply well-typed. In the proof of external completeness we will argue, that the substitution we construct is indeed in the semantics of the standard representation, i.e. makes the root deeply well-typed. In this proof (theorem 4.63), it will be sufficient if we use a solution of a node which makes all the expressions in *that node* deeply well-typed. The deep well-typedness of that lower node will follow from the deep well-typedness of the root and the requirement that only rules are on the path to that node, which propagate the deep well-typedness.

Definition 4.27. Let R be a rule, N and M nodes, where N is a direct successor of M when R is applied to M. Let $n_1, \dots, n_k$ be the expressions in N and $m_1, \dots m_l$ be the expressions in M and let π be the edge connecting M and N. Furthermore, let $\theta\pi$ be a solution for N and θ one for M. R is said to *propagate* $WT_{\Downarrow}$ iff $\forall i : \theta\pi m_i \in WT_{\Downarrow} \implies \forall j : \theta n_j \in WT_{\Downarrow}$.

A tableau's tree is a data structure for the calculi to operate on. It is also a representation of its roots solutions. A more concise representation for the solutions is the *standard representation.* To justify the alternative representation it will be necessary to show that every solution represented by the standard representation is indeed a solution for the tableau's root. Furthermore, it will be desirable that every solution represented by the tableau is represented in the standard representation as well. We will later prove this property (external completeness) for the part of the concretization of the standard representation of a closed tableau making the root deeply well-typed.
We could define the standard representation as:

> The semantics of a standard representation $N = S$ is that part of its concretization which is deeply well-typed, i.e. $\gamma(N) \cap WT_{\Downarrow}$.

But we do not define it this way, because this definition would require substitutions for root variables to be deeply well-typed, even if these variables are projected away. This would, in our opinion, make the definition to strict.
An example will illustrate this:

Example 4.28. The constraint $\texttt{K}\ x\ y \in \texttt{Bot}$ becomes solved with the standard representation $N = [\texttt{Bot}, \texttt{Top}]$, i.e. $\theta \in \gamma(N)_{\mathcal{T}_\mathcal{V}} \implies \theta(\texttt{K}\ x\ y) \in \gamma(\texttt{Bot}) = \eta(\texttt{Bot})$. To spell it out $\theta \in \gamma(N)_{\mathcal{T}_\mathcal{V}} \implies \theta x \in \gamma(\texttt{Bot})$ and $\theta y \in \gamma(\texttt{Top})$ according to lemma 3.43. We can observe that any $\theta \in \gamma(N)_{\mathcal{T}_\mathcal{V}} \implies \theta x \in \gamma(\texttt{Bot})$ for which $\theta x \in WT_{\Downarrow}$, but possibly $\theta y \notin WT_{\Downarrow}$ satisfies $\theta(\texttt{K}\ x\ y) \in WT_{\Downarrow}$ by invariance of $WT_{\Downarrow}$ (since $\texttt{K}\ x\ y \xrightarrow{*}_{\text{no}} x$). So requiring $\theta \in \gamma(N)_{\mathcal{T}_\mathcal{V}}$ to satisfy $\theta y \in WT_{\Downarrow}$ is not necessary to satisfy $\theta(\texttt{K}\ x\ y) \in WT_{\Downarrow}$ which would allow a θ with $\theta y \notin WT_{\Downarrow}$.

Instead, we relate the standard representation to the root of a tableau and only the substitutions from its concretization which map the root to an element of $WT_{\Downarrow}$ are counted as belonging to its semantics. To put it another way: the semantics of the standard representation forms a set included in its concretization.

Definition 4.29 ($\gamma^{WT_{\Downarrow}}(\cdot)$, semantics of the standard representation). The *semantics of a standard representation* $N = S$ with respect to a tableau having root $R = \{r_1 \in C_1, \dots, r_n \in C_n\}$ is that part of its concretization which makes R deeply well-typed, i.e. $\gamma^{WT_{\Downarrow}}(N) \overset{\text{def}}{=} \{s \in \gamma(N) | s_{\mathcal{T}_\mathcal{V}} r_i \in WT_{\Downarrow}\}$.

Definition and Corollary 4.30. *Let $\mathcal{T}$ be a tableau with standard representation $\mathcal{T}_N$. We say the* semantics of N *is* $\texttt{bot}$*-closed, or alternatively, N is c-*$\texttt{bot}$*-closed, iff*

$$\forall \mathbb{C}[\cdot], s \in \Lambda : \mathbb{C}[s] \in \gamma^{WT_{\Downarrow}}(N) \implies \mathbb{C}[\texttt{bot}] \in \gamma^{WT_{\Downarrow}}(N).$$

If N is $\texttt{bot}$*-closed then N is c-*$\texttt{bot}$*-closed.*

Proof. This is a corollary of lemma 4.20. □

Definition 4.31 (internally sound tableau, internally complete tableau). A tableau is *internally sound*, if for every internal node N with successors $N_1, \ldots, N_n$ reachable by edges labeled $\pi_1, \ldots, \pi_n$, respectively, the inclusion $U(N_i)\pi_i \subseteq U(N)$ holds, it is *internally complete*, if $U^c_{\mathcal{T}}(N) \subseteq \bigcup_i U^c_{\mathcal{T}}(N_i)\pi_i$.

If we can prove the condition for completeness for all solutions including the ones making the root expression ill-typed, that will of course imply completeness.

Lemma 4.32. $U(N) \subseteq \bigcup_i U(N_i) \implies U^c_{\mathcal{T}}(N) \subseteq \bigcup_i U^c_{\mathcal{T}}(N_i)$.

Definition 4.33 (sound rule, complete rule). Rules transforming a sound tableau to a sound tableau are termed *sound rules* and those transforming a complete tableau to a complete tableau are termed *complete rules*.

Lemma 4.34. *Some very basic criteria for sound and complete rules are:*

1. *Every rule which only removes constraints is complete.*
2. *Every rule which only adds constraints is sound.*
3. *Every rule adding only a leaf labeled* no! *is sound.*

Proof. We only show the first of these criteria, the other proofs are similar.
Assume the rule adds N below M connected by edge π. Identifying N with its label, we can write $M = \{s_1 \in C_1, \ldots, s_n \in C_n\} \cup N$. Any solution $\theta \in U^c_{\mathcal{T}}(M)$ will also satisfy $\theta \in U^c_{\mathcal{T}}(N)$, since both nodes relate to the same root. □

We have to relate the result of the calculi to its inputs. In order to differentiate this relation from the relation of a node and its direct descendants, we use the attribute external for the former

and internal for the latter. For external soundness we need to show that every substitution contained in the result is indeed a solution of the input. For external completeness we need to show analogously that every c-solution of the input is indeed contained in the result of the calculus.
We begin by formalizing the notions of external soundness and external completeness.

Definition 4.35 (externally sound, externally complete). A calculus is *externally* (*extensionally* or *globally*) *sound*, if for any tableau, $\mathcal{T}$, closed with the calculus and having root $R = \{\mathbb{C}[x_1, \ldots, x_n] \in D\}$ and standard representation $\mathcal{T}_N : \gamma(N)_{\mathcal{T}_\mathcal{V}} \subseteq U(R)$. The calculus is *externally* (*extensionally* or *globally*) *complete*, if $\gamma^{WT_\Downarrow}(N)_{\mathcal{T}_\mathcal{V}} \supseteq U^c_{\mathcal{T}}(R)$.

It is important enough to reiterate that soundness as well as completeness are defined with respect to a closed tableau and are undefined for tableaus which are not closed.
We introduce some notation for the function of the calculi.

Definition 4.36 ($\to_{\mathsf{ADE}}$, $\to_{\mathsf{CADE}}$). Let $\mathcal{T}$ be a demand tableau closed with ADE, having root $R = \{t_1 \in C_1, \ldots, t_n \in C_n\}$, standard representation $\mathcal{T}_N$ and root variables $\mathcal{T}_\mathcal{V}$ then we say $\{t_1 \in C_1, \ldots, t_n \in C_n\}$ *demands* $\mathcal{T}_\mathcal{V} \in N$ and write

$$\{t_1 \in C_1, \ldots, t_n \in C_n\} \to_{\text{ADE}} N.$$

If $\mathcal{T}$ was closed with CADE we say *demands only* and write $\to_{\text{CADE}}$ instead of *demands* and $\to_{\text{ADE}}$, respectively.

Remark 4.37. Obviously, $\to_{\text{CADE}} \subseteq \to_{\text{ADE}}$.

4.3 Rules

4.4 Local Rules

Lemma 4.38. *A rule with premises implying $s \equiv_c t$ and $\mathcal{CV}(s) = \mathcal{CV}(t)$ and*

$$\frac{\{s \in C\} \cup R}{\{t \in C\} \cup R} \tag{4.1}$$

is sound and complete.

Proof. Lemma 2.100 implies $s \equiv_c t \iff \forall\sigma : \sigma s, \sigma t \in \Lambda^0 \implies \sigma s \leq_c \sigma t$. Thus for every substitution θ with $\theta t \in \gamma(C)$ the $\equiv_c$-closure of $\gamma(C)$ implies $\theta t \in \gamma(C)$ and vice versa. This implies soundness and completeness. □

Likewise, every rule which replaces a demand with a $\equiv_\gamma$-equivalent demand is sound and complete. Examples of such rules are moving unions up towards the root of the demand expression or replacing the name of a demand with its definition. We give a name to the latter since it is quite commonly used.

$$\frac{\{s \in N\} \cup R}{\{s \in D\} \cup R}, \text{ if } N = D \text{ is a demand definition} \tag{name}$$

4.4.1 Redundant `Top`

If the right hand side in a constraint is `Top`, we may remove that constraint.

$$\frac{\{s \in \texttt{Top}\} \cup R}{R} \tag{redtop}$$

Theorem 4.39. *The rule for removing redundant* `Top` *is sound and complete.*

Proof.

sound: Let $\mathcal{T}^1$ be a sound tableau with a leaf $M = \{s \in \texttt{Top}\} \cup R$ and let $\mathcal{T}^2$ be the tableau resulting from application of (redtop) to this leaf. For $\mathcal{T}^2$ to be a sound tableau for every successor N (that is only the node R in this case) $U(N)\pi \subseteq U(M)$ must hold where π labels the edge from M to N (in this case $\pi = id$). Every solution $\theta \in U(N)$ must substitute all constrained variables with ground expressions, in particular all variables occurring free in s. With proposition 3.98 and $\theta s \in \Lambda^0$ we obtain $\theta s \in \gamma(\texttt{Top})$. Obviously, θ solves the remaining constraints R, so that θ is also a solution for M.

complete: We need to show $U^c_{\mathcal{T}}(M) \subseteq U^c_{\mathcal{T}}(N)$. Whatever $\theta \in U^c_{\mathcal{T}}(M)$ substitutes for the free variables in s it will be in $\gamma(\texttt{Top})$ and will furthermore make the root deeply well-typed, so it will be a c-solution for N. □

4.4.2 Reduction

In every constraint the left hand side is a Λ-expression. The tableau may be extended by applying normal-order reduction to this left hand side of a constraint.

$$\frac{\{s \in A\} \cup R}{\{t \in A\} \cup R}, \text{ if } s \rightarrow_{\text{no}} t \tag{red}$$

Theorem 4.40. *The normal-order reduction rule is sound and complete.*

Proof. Theorem 2.93 and the absence of an edge label provide the prerequisites for applying lemma 4.38 which directly implies the statement. □

4.4.3 Constructors and Jocs

If the Λ-expression in a constraint has constructor `c` as its top most symbol and if the demand in that constraint is of the form $\mathtt{c}\ C_1 \dots C_{\alpha(\mathtt{c})}$ the analysis can proceed to the arguments of `c`.

$$\frac{\{\mathtt{c}\ t_1 \dots t_{\alpha(\mathtt{c})} \in \mathtt{c}\ C_1 \dots C_{\alpha(\mathtt{c})}\} \cup R}{\{t_1 \in C_1, \dots, t_{\alpha(\mathtt{c})} \in C_{\alpha(\mathtt{c})}\} \cup R} \qquad \text{(decomp)}$$

Theorem 4.41. *Constructor decomposition is sound and complete.*

Proof. The set of solutions for both nodes (predecessor and descendant) remains unchanged. This suffices for soundness. For completeness we remark that both nodes relate to the same root, so the set of c-solutions is the same. □

If there are no Λ-expressions represented by some constraints demand, which start with the same constructor as that constraints Λ-expression we add an additional label *no!*.

$$\frac{\{\mathtt{c}\ t_1 \dots t_{\alpha(\mathtt{c})} \in A\} \cup R}{\mathit{no!}\ \{\mathtt{c}\ t_1 \dots t_{\alpha(\mathtt{c})} \in A\} \cup R}, \text{ if } \gamma(A) \cap \gamma(\mathtt{c\ Top \dots Top}) = \emptyset \quad \text{(wc)}$$

Theorem 4.42. *The rule for mismatched constructors is sound and complete.*

Proof. Soundness is a direct consequence of lemma 4.34. $\gamma(A) \cap \gamma(\mathtt{c\ Top \dots Top}) = \emptyset$ implies that for any substitution $\theta : \theta(\mathtt{c\ Top \dots Top}) \notin \gamma(A)$ which entails completeness. □

While the rule for mismatched constructors is sound and complete it is also undecidable to test whether it is applicable. An implementation will use some decidable conditions for emptiness of $\gamma(A) \cap \gamma(\mathtt{c\ Top \dots Top})$, such as $A \equiv \mathtt{c}'\ A_1 \dots A_{\alpha(\mathtt{c}')}$ where $\mathtt{c} \neq \mathtt{c}'$, $A \equiv \mathtt{Fun}$ or $A \equiv \mathtt{Bot}$.

Joc

Decomposition

The rule for joc decomposition is in fact an instance of the rule for constructor decomposition:

$$\frac{\{\mathtt{[}t_1, \dots, t_n\mathtt{]} \in \mathtt{[}C_1, \dots, C_n\mathtt{]}\} \cup R}{\{t_1 \in C_1, \dots, t_n \in C_n\} \cup R} \qquad \text{(jocdec)}$$

Soundness and completeness follows from soundness and completeness of constructor decomposition.

4.4.4 Rules for `bot`

If `bot` is encountered in a reduction context $\mathbb{R}[\cdot]$ we can identify the entire expression with `bot`.

$$\frac{\{\mathbb{R}[\mathtt{bot}] \in C\} \cup R}{\{\mathtt{bot} \in C\} \cup R} \qquad \text{(absred)}$$

Theorem 4.43. *The* (absred) *is sound and complete.*

Proof. The criterion of lemma 4.38 is satisfied. □

The expression `bot` is a representative of `Bot` in Λ. Therefore $\mathtt{bot} \in \mathtt{Bot}$ holds.

$$\frac{\{\texttt{bot} \in \texttt{Bot}\} \cup R}{R} \qquad \text{(redbot)}$$

Theorem 4.44. *The rule for redundant* `bot` *is sound and complete.*

Proof.

complete: One constraint is removed thus by lemma 4.34 this rule is complete.

sound: $\theta\texttt{bot} \in \gamma(\texttt{Bot})$ holds for every substitution θ, therefore every solution is a solution of $\texttt{bot} \in \texttt{Bot}$. Hence every solution of R is a solution of $\{\texttt{bot} \in \texttt{Bot}\} \cup R$. □

Since `bot` is not a member of every representation of a demand some of the constraints with `bot` do not have any solution.

$$\frac{\{\texttt{bot} \in A\} \cup R}{\textit{no!}\ \{\texttt{bot} \in A\} \cup R}, \text{ if } \texttt{Bot} \not\leq_\gamma A \qquad \text{(nobot)}$$

Theorem 4.45. *The rule for mismatched* `bot` *is sound and complete.*

Proof.

complete: There cannot be any substitution θ for which $\theta\texttt{bot} \in \gamma(A)$, because otherwise for any diverging $t : t \in \gamma(A)$ would have to hold, but this would imply $\gamma(\texttt{Bot}) \subseteq \gamma(A)$.

sound: By lemma 4.34 every rule adding *no!* is sound. □

4.4.5 Abstraction

Redundant `Fun`

$$\frac{\{s \in \texttt{Fun}\} \cup R}{R}, \text{ if } s \text{ is an FWHNF} \qquad \text{(redfun)}$$

Theorem 4.46. *The rule* (redfun) *is sound and complete.*

Proof.

sound: Since WHNFs and in particular FWHNFs are stable with respect to arbitrary substitution, every substitution θ that solves R solves $\{s \in \texttt{Fun}\} \cup R$.

complete: We can apply lemma 4.34. □

Conflicting `Fun`

Some Λ-expressions are easily seen not to belong to `Fun`.

$$\frac{\{s \in \texttt{Fun}\} \cup R}{\textit{no!}\ \{s \in \texttt{Fun}\} \cup R}, \text{ if } s \equiv \texttt{bot} \vee s \equiv \texttt{c}\ s_1 \ldots s_{\alpha(\texttt{c})} \qquad \text{(nofun)}$$

Theorem 4.47. *The rule* (nofun) *is sound and complete.*

Proof.

sound: Soundness is an immediate consequence of lemma 4.34.

complete: From proposition 2.35 we conclude that under no substitution `bot` or $\texttt{c}\ s_1 \ldots s_{\alpha(\texttt{c})}$ become expressions having an FWHNF. Thus there is no substitution θ with $\theta s \in \gamma(\texttt{Fun})$. □

4.4.6 Type error

By hypothesis Λ is well-typed. We only need solutions making the root deeply well-typed, thus we can extend a tree with a leaf containing an ill-typed expression with a leaf without any solution.

$$\frac{\{s \in A\} \cup R}{\textit{no!}\ \{s \in A\} \cup R}, \text{ if } s \in IT \tag{type}$$

Theorem 4.48. *The rule for type errors is sound.*

Proof. Follows immediately from lemma 4.34. □

In general, (type) is incomplete. We will postpone the discussion of this fact until all local rules have been presented (cf. lemma 4.59), since (type)s incompleteness depends on the presence of other local rules on the path from the root to the leaf.

4.4.7 Union and intersection

If an expression is constrained by a union we can compose the solution for this constraint from the solutions of the constraints made with the union's components.

$$\frac{\{s \in \langle C_1, \ldots, C_n\rangle\} \cup R}{\{s \in C_1\} \cup R \mid \ldots \mid \{s \in C_n\} \cup R} \tag{union}$$

Theorem 4.49. *The rule decomposing unions is sound and complete.*

Proof. The set of solutions remains the same, since by lemma 3.43 we have $\theta s \in \gamma(\langle C_1, \ldots, C_n\rangle) \iff \theta s \in \bigcup_i \gamma(C_i)$. □

If a single Λ-expression has to satisfy multiple constraints we may combine these.

$$\frac{\{s \in C, s \in D\} \cup R}{\{s \in C \cap D\} \cup R} \tag{is}$$

Theorem 4.50. *The rule* (is) *is sound but not necessarily complete.*

Proof.

sound: From lemma 3.43 we know that $\gamma(C \cap D) \subseteq \gamma(C) \cap \gamma(D)$ implying that any solution of the descendant is a solution of the predecessor.

incomplete: In the same lemma we present an example showing $\gamma(C) \cap \gamma(D) \notin \gamma(C \cap D)$. □

Turning (is) around we could decompose an intersection into multiple constraints for the same expression. But since this rule would not even be sound, we do not provide it in the calculi.

4.4.8 Case

We have already covered the case where a $\texttt{case}_A$ is applied to an expression in CWHNF with the reduction rule.
Here we give a rule which can be applied provided that the $\texttt{case}_A$ expression is a potential redex. In that case we can insert the constructor expressions for the variable. Note that substitutions might substitute the left hand side of VCs and thus transform VCs back into general constraints.
The ρ_i substitute the ith constructor of A and ρ_0 substitutes `bot` for the cased variable, x, i.e. let $x_{i,j}$ be fresh variables and

let $\rho_0 = \{x \mapsto \texttt{bot}\}, \rho_i = \{x \mapsto \texttt{c}_{A,i}\ x_{i,1} \dots x_{i,\alpha(\texttt{c}_{A,i})}\}, L_i = \rho_i(\{\mathbb{R}[\texttt{case}_A\ x\ e_1 \dots e_{|A|}] \in C\} \cup R)$ and $R = \{s_i \in C_i | 1 \leq i \leq n\}$. We extend a tableau with

$$\frac{\{\mathbb{R}[\texttt{case}_A\ x\ e_1 \dots e_{|A|}] \in C\} \cup R}{L_0 \mid \dots \mid L_{|A|}} \qquad \text{(casep)}$$

if the $\texttt{case}_A\ x\ \dots$ expression is a potential redex, i.e. if its first argument is a variable. Every edge is labeled with the corresponding ρ_i.

Theorem 4.51. *The* $\texttt{case}_A$ *rule for potential redexes is sound and complete.*

Proof.

sound: Let θ be a solution for one of the L_i. Then $\theta\mathbb{R}[\texttt{case}_A\ (\texttt{c}_{A,i}\ x_{i,1} \dots x_{i,\alpha(\texttt{c}_{A,i})})\ e_1 \dots e_{|A|}] \in \gamma(C) \wedge \theta \in U(\rho_i R)$ which is equivalent to $\theta\rho_i\mathbb{R}[\texttt{case}_A\ x\ e_1 \dots e_{|A|}] \in \gamma(C) \wedge \theta\rho_i \in U(R)$ and soundness holds.

complete: A c-solution θ of $L = \{\mathbb{R}[\texttt{case}_A\ x\ e_1 \dots e_{|A|}]\} \cup R$ satisfies $\theta\mathbb{R}[\texttt{case}_A\ x\ e_1 \dots e_{|A|}] \in \gamma(C)$, $\theta \in U^c_{\mathcal{T}}(R)$ and with the substitution π along the path from the root to L the substitution $\theta\pi$ must make the root deeply well-typed and thus must substitute x since x cannot be bound by $\mathbb{R}[\cdot]$. Due to definition 4.22 θ must substitute x with an expression having a saturated constructor WHNF with top level constructor $\texttt{c}_{A,i}$ or with $\texttt{bot}$, otherwise the substituted root would not be in $WT_{\Downarrow}$ and θ would not be a c-solution. As a consequence θ can be written as $\theta = \theta_i\rho_i$, thus $U_\eta(L)\rho_i \subseteq \bigcup_i U_\eta(L_i)$ holds implying completeness of the rule. □

4.4.9 Reuse

A prerequisite of the soundness proof which we give for the entire calculi in section 4.6 is that the higher node for an application of the loop rule is the root of the tree. In practice this would be a serious limitation, resulting in many cases for which the tableau could not be closed and would grow infinitely.
In many of these cases there is a “sub-analysis”, which could have been performed independently and, if it had, would have been closed by the loop rule. This is the motivation for the reuse rule, which uses results computed in a prior analysis in the present analysis.
Let $\mathcal{T}^t$ be a closed tableau with root $t \in C$, standard representation $\mathcal{T}^t_N$ and root variables $\mathcal{T}^t_{\mathcal{V}}$. Let τ be a substitution satisfying $\tau t \equiv s$. We may extend the tableau by the reuse rule:

$$\frac{\{s \in C\} \cup R}{\{\tau\mathcal{T}^t_{\mathcal{V}} \in N\} \cup R} \qquad \text{(reuse)}$$

Theorem 4.52. *Reuse is sound.*

Proof. We will suppose the result of analyzing $t \in C$ to be sound. Consider a solution θ of $\tau\mathcal{T}^t_{\mathcal{V}} \in N$. By definition $\theta\tau\mathcal{T}^t_{\mathcal{V}} \in \gamma(N)$ and it follows that $\theta\tau\mathcal{T}^t_{\mathcal{V}_{\mathcal{T}^t_{\mathcal{V}}}} t \equiv \theta\tau t \equiv \theta s \in \gamma(C)$. □

It seems that (reuse) is incomplete, because even if the analysis reused, $t \in C$, is itself complete, this completeness refers to the subset of the result, N, making t deeply well-typed. This last restriction however is lost by the (reuse) rule. It seems possible for solutions of $t \in C$ not making t deeply well-typed to be missing from $\gamma(N)$. It might also be possible to argue in favor of (reuse)’s completeness considering the deep well-typedness of the root for the reusing tableau. We do not investigate this here, but leave it to future work.

4.4.10 Local rule properties

We will present some properties of the local rules, which will later be used in the proofs of external soundness and external completeness.
The first observation concerns the variables which may appear in VCs. A variable may either appear in a VC or there is a substitution for it along the path from the root to the node, but not both.
Obviously, variables can occur freely in the input constraint or they can be introduced by the rule (casep).

Lemma 4.53. *Let σ be the substitution along the path from the root to a node N. Either $x_i \in \mathcal{CV}(N)$ or $\sigma x_i \neq x_i$.*

Proof. The (partial) substitutions along the path are introduced on behalf of the rule (casep). They are applied to the entire node and introduce fresh variables only. So if a variable is substituted it will no longer be constrained and as long as it is constrained it cannot have been substituted. □

Lemma 4.54. *Rules along a narrow path except reuse propagate $WT_{\Downarrow}$.*

Proof. Let $\mathcal{P}$ be the narrow path from constraint $r \in A$ in M to constraint $s \in B$ in N. We consider each of the different rules which may appear on $\mathcal{P}$.

(casep) The `case` to which the rule is applied may be within r or may be in some other constraint. In both cases deep

well-typedness is propagated: since the variable being cased, say x, will occur free in M but not in N, we can write any solution $\tau \in U(M)$ as $\theta\{x \mapsto \dots\}$ and if $\forall i : \theta\{x \mapsto \dots\}m_i \in WT_{\Downarrow}$ then $\forall i : \theta n_i \in WT_{\Downarrow}$ since $\{x \mapsto \dots\}m_i \equiv n_i$.

(red) The constraint in the upper node is $r \in A$ and $r \rightarrow_{\text{no}} t$. If θr has a WHNF r', then due to invariance of termination there is a t' which is a WHNF of θr and θt. Since θ is a ground substitution for all free variables in r and t, t' is a closed WHNF. t' may be an SCWHNF or an FWHNF. In both cases $t \in WT_{\Downarrow}$, since r and thus t' are deeply well-typed. If θr has no WHNF, $\theta r \in WT_{\Downarrow}$ implies $\theta r \in WT$. Thus we cannot reach an expression in DIT by normal-order reduction and $\theta t \in WT$ and also in $WT_{\Downarrow}$.

(decomp) From lemma 4.17 we know that θ makes the constructor arguments deeply well-typed.

(jocdec) as for decomp.

(reuse) The reuse rule may not propagate $WT_{\Downarrow}$. A node with the constraint $\texttt{K}\ x\ y \in \texttt{Bot}$ may get a successor with $[x, y] \in N$, where $N = [\texttt{Bot}, \texttt{Top}]$ is the result of analyzing $\texttt{K}\ x\ y \in \texttt{Bot}$. Replacing N we obtain $[x, y] \in [\texttt{Bot}, \texttt{Top}]$. But $\theta(\texttt{K}\ x\ y) \in WT_{\Downarrow}$ does not imply $\theta y \in WT_{\Downarrow}$. □

For discussions involving tableaus using the (reuse) rule it will be helpful to know that externally sound and complete analyses cast the `bot`-closedness of their demand to the analysis result.

Lemma 4.55. *Let $\mathcal{T}$ be a closed tableau for input $\mathbb{C}[\vec{x}] \in D$, let $\mathcal{T}$ be externally sound and complete with standard representation $\mathcal{T}_N$,* *and let D be* `bot`*-closed and deeply well-typed, then the semantics of N is also* `bot`*-closed.*

Proof. $\forall \mathbb{D}[\cdot], s : \mathbb{D}[s] \in \gamma^{WT_\Downarrow}(N) \iff \mathbb{C}[\mathbb{D}[s]] \in \gamma(D) \cap WT_\Downarrow$ since N is a sound and complete solution.
Assume N's semantics is not `bot`-closed. We can find $\mathbb{D}[s]$ with $\mathbb{D}[s] \in \gamma^{WT_\Downarrow}(N)$ but $\mathbb{D}[\texttt{bot}] \notin \gamma^{WT_\Downarrow}(N)$. It follows that $\mathbb{C}[\mathbb{D}[s]] \in \gamma(D) \cap WT_\Downarrow$ but $\mathbb{C}[\mathbb{D}[\texttt{bot}]] \notin \gamma(D) \cap WT_\Downarrow$ in contradiction to the `bot`-closedness of D. We conclude that N's semantics must be `bot`-closed. □

Remark 4.56. The premises in lemma 4.55 are not sufficient to conclude that N is `bot`-closed. An analysis of $\mathbb{C}[x] \in D$ could completely demand $x \in \langle \texttt{Bot}, \texttt{c}_1\ \texttt{Bot}, \texttt{c}_2\ \texttt{Fun}\rangle$. Yet, it could be that $\forall s \in \gamma(\texttt{Fun}) : \mathbb{C}[\texttt{c}_2\ s] \notin WT_\Downarrow$, so N would not be `bot`-closed but it would be c-`bot`-closed.

Remark 4.57. Let $\mathbb{C}[\vec{x}] \in D \rightarrow_{\text{CADE}} N$ and let $\texttt{Bot} \leq_\gamma D$, then $\texttt{Bot} \leq_\gamma N$ is not necessarily true.

Proof. We provide an example.
Let $\mathbb{C}[\cdot] \equiv \texttt{True} : [\cdot]$, and let $D = \langle \texttt{Bot}, \texttt{Top} : \texttt{Top} : \texttt{Bot}\rangle$ be D's demand definition. The analysis results in $N = \texttt{Top} : \texttt{Bot}$ and is sound and complete, but $\texttt{Bot} \not\leq N$. □

Lemma 4.58. *Let $\mathcal{T}_T$ be a tableau's tree and let $\mathcal{P}$ be a path in this tree from a node M to a node N. Furthermore let σ be the substitution along $\mathcal{P}$ and let $\overrightarrow{s}$ and $\overrightarrow{t}$ be Λ-expressions at M and N respectively. If on a narrow path $\mathcal{Q}$ from the constraint $s_i \in D_i$ to the constraint $t_j \in C_j$ there is no* (reuse) *rule, then*

$$\exists \pi \in \Pi(\sigma s_i) : (\sigma s_i)_{|\pi} \equiv_c t_j.$$

Proof. The only rules along $\mathcal{P}$ which can change the Λ-expression s_i are (casep), (decomp), (absred) and (red) rules along $\mathcal{Q}$.
We prove the statement with induction on the sequence of the rules along $\mathcal{Q}$. Let q be the sequence consisting of the (casep), (decomp), (absred) and (red) rules along $\mathcal{Q}$.

$q = \varepsilon$: Since (casep) is the only rule introducing a non-trivial edge $\sigma s_i \equiv s_i$ and since no rule except the above affects s_i we get $s_i \equiv t_j$ and the statement is proved.

$q = (q_1, q_2, \dots, q_n)$: We distinguish the different cases for q_1.

- $q_1 =$ (red) : $s_i \rightarrow_{\text{no}} s'_i$ and we can apply the induction hypothesis to $(q_2, \dots, q_n)$. It follows that $\exists \pi \in \Pi(\sigma s'_i) : (\sigma s'_i)_{|\pi} \equiv_c t_j$. But s_i and s'_i have exactly the same primitive positions (they are $\equiv_c$-equivalent) and thus $\exists \pi \in \Pi(\sigma s_i) : (\sigma s_i)_{|\pi} \equiv_c t_j$.
- $q_1 =$ (decomp) : $s_i \equiv \texttt{c}\ r_1 \dots r_{\alpha(\texttt{c})}$ and $\mathcal{P}$ goes to the successor using the kth argument, i.e. to the node $\{r_k \in D_i^k\} \cup R$. From the induction hypothesis we obtain $\exists \pi \in \Pi(\sigma r_k) : (\sigma r_k)_{|\pi} \equiv_c t_j$. Then obviously $\exists \pi \in \Pi(\sigma s_i) : (\sigma s_i)_{|\pi} \equiv_c t_j$ holds.
- $q_1 =$ (casep) : Here we may have two cases.
  - $q_1 \in \mathcal{Q}$: $s_i \equiv \texttt{case}_A\ x\ \overrightarrow{e}$ and $\mathcal{P}$ goes to the successor along the edge $\rho = \{x \mapsto r\}$. This node is labeled $\{\texttt{case}_A\ r\ \rho\overrightarrow{e} \in D_i\} \cup \rho R$. We may write $\sigma = \sigma'\rho$. Again we can apply the induction hypothesis and obtain $\exists \pi \in \Pi(\sigma'(\texttt{case}_A\ r\ \rho\vec{e})) : (\sigma'(\texttt{case}_A\ r\ \rho\vec{e}))_{|\pi} \equiv_c t_j$, but $\sigma'(\texttt{case}_A\ r\ \rho\vec{e}) \equiv \sigma s_i$.

$q_1 \notin \mathcal{Q}$: $\mathcal{P}$ goes to the successor along $\rho = \{x \mapsto r\}$. There the expression in place of s_i is ρs_i. We write $\sigma = \sigma'\rho$ and apply the induction hypothesis to obtain $\exists \pi \in \Pi(\sigma'(\rho s_i)) : (\sigma'(\rho s_i))_{|\pi} \equiv_c t_j$, but $\sigma'(\rho s_i) \equiv \sigma s_i$.

$q_1 =$ (absred) : $s_i \equiv \mathbb{R}[\texttt{bot}]$ and (absred) replaces s_i with `bot`. $\mathbb{R}[\texttt{bot}] \equiv_c$ `bot` so the statement holds with $\pi = \lambda$. In this case the induction hypothesis provides for $t_j \equiv$ `bot`. □

Proposition 4.59. *The* (type) *rule may be incomplete, if a* (reuse) *rule is present on the narrow path from the root to the node.*

Proof. We provide an example. Assume we had already analysed `K` x y $\in$ `Bot` and had obtained [`Bot`, `Top`] as result. If we analyse `K bot (Nil` z`)` $\in$ `Bot` the following tree can be produced: But since `K bot (Nil` z`)` $\to_{\text{no}}$ `bot` for any substitution θ : $\theta(\texttt{K bot (Nil } z\texttt{)}) \in \gamma(\texttt{Bot})$ is satisfied and so is `K bot (Nil` z`)` $\in WT_{\Downarrow}$. Thus (type) is incomplete. □

$$\begin{tikzcd}
\texttt{K bot ([] } z\texttt{)} \in \texttt{Bot} \arrow[d, dash, "\text{(reuse)}"] \\
[\texttt{Bot}, (\texttt{[] } z)] \in [\texttt{Bot}, \texttt{Top}] \arrow[d, dash, "\text{(decomp)}"] \\
\texttt{bot} \in \texttt{Bot}, \texttt{[] } z \in \texttt{Top} \arrow[d, dash, "\text{(type)}"] \\
\ldots \textit{no!}
\end{tikzcd}$$

Figure 4.2: Incomplete application of (reuse)

Intuitively, the reason is that the definition of $WT_{\Downarrow}$ uses only *needed* positions of an expression and recursively of its SCWHNF's arguments. (type) might be extended allowing its application below (reuse) if the constraint to which it is applied stems from a needed position of the (reuse) rule, but we do not investigate this any further.

4.5 Global rules

Global rules have a wider scope than local rules: in order to determine their applicability the immediate predecessor is not sufficient and their prerequisites require conditions further up in the path from the root to hold.

In an analysis we might arrive at a leaf which is similar to a leaf we have seen earlier in that analysis. Here the labels are considered similar if the root's expression is equivalent to a multi-context $\mathbb{C}[\cdots]$ with the root variables $\mathcal{T}_{\mathcal{V}}$ in the holes and every Λ-expression in the leaf's label can be formed by inserting some expressions into $\mathbb{C}[\cdots]$'s holes. We call this situation a *potential loop*, because it might be possible to similarly repeat all of the tableau below this leaf.

The solutions to the tableau will then depend on some observations on the path from the root to the leaf.

We can distinguish the following cases:

1. The Λ-expressions in the leaf in place of the root variables have become smaller than what was assumed of the root variables structure along the path. Then, if our metric is

well founded, we will obtain a finite tableau for any finite input formed by repeating this structure. This amounts to a particular induction proof.

2. The expressions in place of the root variables are exactly the same as what was assumed of these variables on the path to the leaf.

   a) If a constructor decomposition has occurred on the path from the root, and if we know that we will need to compute the Λ-expression to arbitrary precision, thus being able to repeat that same constructor decomposition again and again, the constraint will not contribute to the solution, because, intuitively, this process may continue forever and all the finite solutions are already solutions of the tableau above the leaf. So the constraint is dropped from the leaf.

   b) If no constructor decomposition has occurred but at least one abstract reduction, which was called for by a `Bot` present in the demand, we have an analogous case to 2a: the process on the constraint may go on infinitely without resulting in solutions not already present in the tableau above the leaf. As in 2a the constraint is dropped from the leaf.

   c) If the situation is as for 2b but the reduction was not called for by a `Bot` in the demand, we have detected an infinite computation for the input assumed on this path, but only finite computations were demanded, thus there are no solutions to the leaf and we mark the leaf *no!*.

Equipped with the intuition above we now present the global rules more formally. A potential loop is always detected between a leaf and the root. The rule for loop detection seems to be severely restricted by the requirement, that the top node must always be the root. In conjunction with (reuse) the effect is much softer and merely causes all loops in sub-tableaux to use the same top node. This, admittedly, is still a restriction not all tableaux will meet, but it is far less severe than it at first seems. Further investigation of just how severe this restriction is will be left to future research.
Let *size* be a real-valued, well-founded measure on patterns compatible with the term structure. The size of a substitution is the sum of the sizes of the substituted expressions.
Let $\mathbb{C}[\cdot_1, \ldots, \cdot_n]$ be n-ary context, where $\{\mathbb{C}[x_1, \ldots, x_n] \in D\}$ is the label at the root. Let $\{\mathbb{C}[t_1^1, \ldots, t_n^1] \in D, \ldots, \mathbb{C}[t_1^m, \ldots, t_n^m] \in D\} \cup R$ be the label at the leaf. Furthermore, let R be a VC. Let $\mathcal{P}$ be the path from the root to the leaf, σ be the substitution along $\mathcal{P}$ and σ_R be the substitution which for each $x_i \in C_i$ in R substitutes $x_i \mapsto x_i \cap C_i$ and let $\top$ be the substitution which substitutes `Top` for all the constrained variables not occurring free in R nor in any $\vec{t^j}$. We consider five different global rules:
Assume the following conditions are satisfied:

· the expressions t_i^j consist only of variables, constructors and `bot`.

· $\forall j : \mathcal{FV}(\vec{t^j}) \cap \{x | (x \in D) \in R\} = \emptyset$

$$\frac{\{\mathbb{C}[t_1^1, \ldots, t_n^1] \in D, \ldots, \mathbb{C}[t_1^m, \ldots, t_n^m] \in D\} \cup R}{\textit{loop!}\ \top\sigma_R\sigma\vec{x}\ \texttt{where}\ [[t_1^1, \ldots, t_n^1], \ldots, [t_1^m, \ldots, t_n^m]] = \underbrace{[N, \ldots, N]}_{m}}$$

(loop)

where $\mathcal{T}_N$ is the standard representation for the current tableau. (c-loop) only differs from (loop) in the additional requirements

- for all j: $\mathit{size}(\sigma\vec{x}) > \mathit{size}(\vec{t})$
- no (reuse) is used on the narrow path from $\mathbb{C}[\vec{x}] \in D$ at the root to any of the $\mathbb{C}[\vec{t^i}] \in D$ at the leaf
- the free variables are used at least as often in $\sigma\vec{x}$ as in $\vec{t^j}$

$$\frac{\{\mathbb{C}[t_1^1,\dots,t_n^1] \in D, \dots, \mathbb{C}[t_1^m,\dots,t_n^m] \in D\} \cup R}{\mathit{loop!}\ \top\sigma_R\sigma\vec{x}\ \texttt{where}\ [[t_1^1,\dots,t_n^1],\dots,[t_1^m,\dots,t_n^m]] = \underbrace{[N,\dots,N]}_{m}}$$

(c-loop)

Immediate unions offer a decidable criterion for `bot`, SCWHNFs or FWHNFs selecting at most one component of the union that could contain the expression in its concretization.

Definition 4.60 (immediate union). A union $D \equiv \langle D_1,\dots,D_n\rangle$ is called *immediate*, if there is a decidable criterion, $\tilde{\in}$, such that

$$\begin{aligned}
\texttt{bot}\ \tilde{\in}\ D_i &\implies \texttt{bot} \in \gamma(D) \wedge \forall j \neq i : \texttt{bot} \notin \gamma(D_j) \text{ and}\\
\lambda x.t\ \tilde{\in}\ D_i &\implies \lambda x.t \in \gamma(D_i) \wedge \forall t', j \neq i : \lambda x.t' \notin \gamma(D_j) \text{ and}\\
\texttt{c}\ \vec{s}\ \tilde{\in}\ D_i &\implies \texttt{c}\ \vec{s} \in \gamma(D_i) \wedge \forall \vec{s'}, j \neq i : \texttt{c}\ \vec{s'} \notin \gamma(D_j).
\end{aligned}$$

An example of an immediate union would be a union in which each component starts with a different constructor, possibly below a `where`-expression, or is equivalent to `Bot` or `Fun`.
Assume that for all $j : \sigma\vec{x} \equiv \vec{t^j}$. Since $\forall i,j : \mathbb{C}[\vec{t^i}] \equiv \mathbb{C}[\vec{t^j}]$ and since we use sets of constraints we can assume the leaf to be labeled $\{\mathbb{C}[\vec{t^1}] \in D\} \cup R$ without loss of generality. There are three cases:

1. Let $\mathcal{Q}$ be the narrow path from $\mathbb{C}[\vec{x}] \in D$ in the root to $\mathbb{C}[\vec{t^1}] \in D$ at the leaf and let $\mathcal{Q}_i$ be the narrow paths from $\mathbb{C}[\vec{x}]$ at the root to the components in R at the leaf.
   - If the demand D is `bot`-closed,
   - if R is a VC,
   - and if there is a constructor decomposition along $\mathcal{P}$,
   - if no (reuse) rule occurs on $\mathcal{Q}$,
   - if any union decomposition occurring on $\mathcal{Q}$ decomposes an immediate union demand
   - and if the (reuse) rules possible along the $\mathcal{Q}_i$ are only applied with `bot`-closed demands, then

   $$\frac{\{\mathbb{C}[t_1^1,\dots,t_n^1] \in D\} \cup R}{R} \quad \text{(loopdecomp)}$$

2. If there is no decomposition of a constructor, but $\texttt{Bot} \leq_\gamma D$ there is at least one (red) along $\mathcal{P}$,

   $$\frac{\{\mathbb{C}[t_1^1,\dots,t_n^1] \in D\} \cup R}{R} \quad \text{(loopred)}$$

3. If there is no decomposition of a constructor on top level, but at least one (red) rule along $\mathcal{P}$ and $\texttt{Bot} \not\leq_\gamma D$,

   $$\frac{\{\mathbb{C}[t_1^1,\dots,t_n^1] \in D\} \cup R}{\mathit{no!}\ \{\mathbb{C}[t_1^1,\dots,t_n^1] \in D\} \cup R} \quad \text{(noloop)}$$

Example 4.61. It is necessary that the demand used with the rule (loopdecomp) is `bot`-closed. Otherwise, we can close a tableau with root `repeat` $x \in$ `Fin`, in which we arrive at a node labeled `repeat` $x \in$ `Fin` with rule (loopdecomp). This is not sound since it would represent a solution to `repeat` $x \in$ `Fin` which does not exist. `Fin`, however, is not `bot`-closed since e.g. `bot` $\notin \gamma($`Fin`$)$ and therefore (loopdecomp) is not applicable in this situation.

4.6 External soundness and completeness

Theorem 4.62. *ADE is externally sound.*

Proof. Let $\mathcal{T}$ be a closed tableau with standard representation $\mathcal{T}_N$ defining $N = S$.
We show: if θ is an arbitrary ground substitution of the root variables which is contained in the concretization of the standard representation, then θ is also a solution of the root. In order to be a solution of the root θ has to satisfy $\theta\mathbb{C}[\vec{x}] \in \gamma(D)$. Applying theorem 3.94 it will suffice to show that $\theta\vec{x} \in \eta(N)$ implies that θ solves the root, i.e.

$$\theta\vec{x} \in \eta(N) \implies \theta\mathbb{C}[\vec{x}] \in \gamma(D) \tag{4.2}$$

We proceed by assuming a minimal (with respect to an appropriate measure) counter-example and showing, that there would be a smaller counter-example thus falsifying the assumption and proving our claim.
Assume (4.2) would not hold then $\exists\theta : \theta\vec{x} \in \eta(N) \wedge \theta\mathbb{C}[\vec{x}] \notin \gamma(D)$.
We choose θ to be minimal with respect to the the lexicographic order of the triples consisting of

1. the smallest i, for which $\theta\vec{x} \in \eta_{\Delta^i(\mathbf{0})}(S)$,

2. the number n of normal-order reductions sufficient to reach a WHNF, or ∞ if the expression diverges and

3. the least index k, for which $r_k \notin \eta(D)$ where the $r_i \leq_c r_{i+1}$ form an ascending chain, each of the r_i consists of constructors and `bot` only, their depth increases by exactly one from one to the next and $\bigsqcup_i^c r_i \equiv_c \theta\mathbb{C}[\vec{x}]$, or ∞ if no such chain exists.

The first component is the least significant for the order.
Due to the way the standard representation is generated from $\mathcal{T}$, we can identify the sources of the components that represent θ. I.e. if $S \equiv \langle S_1, \ldots, S_u\rangle$ and $\theta\vec{x} \in \eta(S_j)$ then there is a leaf in $\mathcal{T}_T$ which contributes the jth component to the standard representation. For this leaf we find the following possibilities.

· The leaf is solved without application of the loop rules. The leaf, R, will consist of VCs only and will contribute $\top\dot{\sigma}_R\dot{\sigma}\mathcal{T}_\mathcal{V}$ to the standard representation, so that $\theta = \theta'\sigma_R\sigma$, where σ is the substitution along the path to the leaf, σ_R solves R and $\top$ maps every variable in $\mathcal{CV}(R)$ to `Top`. Due to internal soundness of the tableau $\theta'\sigma_R\sigma$ has to be a solution of the root.

· The leaf was closed with the rule (loop). In this case the proof proceeds as follows. Based on the shape of the standard representation component, S_j, we choose an appropriate substitution of smaller measure than θ represented by N. According to the choice of θ this substitution will then be a solution at the root. Furthermore, we show that a substitution exists, which is a solution of the leaf and which when composed with the substitution along the path gives θ. It follows that θ is a solution at the root.

The component S_j has the form $\top\sigma_R\sigma\vec{x}$ `where` $[\vec{t^1},\ldots,\vec{t^m}] = \underbrace{[N,\ldots,N]}_{m}$. The constraints at the leaf are $\{\mathbb{C}[\vec{t^1}] \in D,\ldots,\mathbb{C}[\vec{t^m}] \in D\} \cup R$ and the constraint at the root is $\mathbb{C}[\vec{x}] \in D$.

We have chosen θ satisfying $\theta \in \eta_{\Delta^i(\mathbf{0})}(S)_{\mathcal{T}_\mathcal{V}}$.

By definition of η, this implies:

$$\begin{aligned}\theta &\in \eta_{\Delta^i(\mathbf{0})}(\sigma_R\sigma\vec{x}\ \texttt{where}\ [\vec{t^1},\ldots,\vec{t^m}] = [N,\ldots,N])_{\mathcal{T}_\mathcal{V}}\\ &= \bigcup_{\rho\in\Sigma(\vec{t^j}):\forall j:\rho\vec{t^j})\sqsubseteq_p^i N} \eta_{\Delta^i(\mathbf{0})}(\rho\sigma_R\sigma\vec{x})_{\mathcal{T}_\mathcal{V}}\end{aligned}$$

Since $\Delta^i(\mathbf{0})(N) = \eta_{\Delta^{i-1}(\mathbf{0})}(S)$

$$= \bigcup_{\rho\in\Sigma(\vec{t^j}):\forall j:\rho\vec{t^j}\sqsubseteq_p^{i-1} S} \eta_{\Delta^i(\mathbf{0})}(\rho\sigma_R\sigma\vec{x})_{\mathcal{T}_\mathcal{V}}$$

We can thus write θ as $\theta = \rho'\rho_R\sigma$ where ρ_R solves the VCs $R = \{y_1 \in D_1,\ldots,y_n \in D_n\}$, i.e. ρ_R does not substitute anything but the y_j and $\rho_R\sigma y_j \in \eta_{\Delta^i(\mathbf{0})}(D_j)$. Because σ is idempotent it will not affect the y_i. For ρ' we demand $\forall j : \rho'\vec{t^j} \in \eta_{\Delta^{i-1}(\mathbf{0})}(S)$. Here we employ the premise that free variables in the $\vec{t^j}$ and in the VCs are disjoint. Since $i-1 < i$ the first component of the measure and thus the measure itself is smaller for $\rho'\vec{t^j}$ than for θ. It follows that $\forall j : \rho'\vec{t^j}_{\mathcal{T}_\mathcal{V}}\mathbb{C}[\vec{x}] \in \gamma(D)$. Since furthermore $\forall j : \rho'\rho_R\vec{t^j} \equiv \rho'\vec{t^j}$ it follows that $\forall j : \rho'\rho_R\vec{t^j}_{\mathcal{T}_\mathcal{V}}\mathbb{C}[\vec{x}] \in \gamma(D)$. $\rho'\rho_R$ is a solution of the leaf, since $\forall j : \rho'\rho_R\mathbb{C}[\vec{t^j}] \equiv \rho'\rho_R\vec{t^j}_{\mathcal{T}_\mathcal{V}}\mathbb{C}[\vec{x}] \in \gamma(D)$.

Since we arrive at the leaf along a path labeled σ, it follows that $\theta = \rho'\rho_R\sigma$ solves the root.

· The leaf was closed with the rule (loopred). Hence there is no constructor decomposition on the path, but a normal-order reduction and `Bot` $\leq_\gamma D$.

The component of the standard representation contributed by the leaf has the form $\top\dot{\sigma}_R\dot{\sigma}\vec{x}$, i.e. we can write θ as $\theta = \theta'\sigma_R\sigma$ where $\dot{\sigma}$ is obtained from σ by substituting `bot` in σ's co-domain with `Bot` (cf. definition 3.11) and where $\sigma_R\sigma y_j \in \eta(D_j)$ for all the VCs $y_j \in D_j$ present in the leaf. The VCs at the leaf are $R = \{y_1 \in D_1,\ldots,y_n \in D_n\}$. In the node above the leaf the constraint $\mathbb{C}[\vec{t^1}] \in D$ is additionally present.

We know that $\theta \in \eta(N)_{\mathcal{T}_\mathcal{V}}$, but $\theta\mathbb{C}[\vec{x}] \notin \gamma(D)$. From `Bot` $\leq D$ and our assumption we infer in particular $\theta\mathbb{C}[\vec{x}] \notin \gamma($`Bot`$) = \eta($`Bot`$)$. So $\theta\mathbb{C}[\vec{x}]$ has a WHNF. The second component of the measure is finite, say n. n normal-order reductions will suffice to obtain a WHNF from $\theta\mathbb{C}[\vec{x}] \equiv \theta'\rho_R\rho\mathbb{C}[\vec{x}]$. The normal-order reductions along the path cannot reduce anything substituted by θ and there is at least one normal-order reduction along the path, hence

$$\theta'\rho_R\rho\mathbb{C}[\vec{x}] \xrightarrow{+}_{\text{no}} \theta'\rho_R\rho\mathbb{C}[\vec{t^1}] \equiv \theta'\rho_R\rho\mathbb{C}[\vec{x}]$$

Thus $n-1$ normal-order reductions suffice to reach the WHNF of $\theta\mathbb{C}[\vec{x}]$. Consider the other components of the measure: since we do not change the substitution, but merely show that for the same substitution fewer normal-order reductions are sufficient, the first and last components of the measure remain unchanged. This, however, contradicts θ's minimality and hence, in this case, there is no such counter-example.

- The leaf was closed with the rule (loopdecomp). Hence D is `bot`-closed and there is a constructor decomposition on the path from the root.

  As above the component of the standard representation contributed by the leaf has the form $\top\dot{\sigma}_R\dot{\sigma}\mathcal{T}_{\mathcal{V}}$ and we can thus write $\theta = \theta'\sigma_R\sigma$.

  The substituted root may be non-primitive and so may have infinitely long primitive positions allowing for infinitely many constructor decompositions. Therefore the argument uses the elements of an ascending chain approximating the substituted root expression.

  We want to contradict the assumption $\theta\mathbb{C}[\vec{x}] \notin \gamma(D)$. Consider an ascending chain $r_1 \leq_c r_2 \leq_c \dots$ of expressions consisting of constructors and `bot` only with $\bigsqcup_i^c r_i \equiv_c \theta\mathbb{C}[\vec{x}]$. If all the $r_i \in \eta(D)$ then $\theta\mathbb{C}[\vec{x}] \in \gamma(D)$, thus there must be a smallest index j for which $\forall j \leq i : r_i \notin \eta(D)$. This holds for all larger indices, because $\eta(D)$ is `bot`-closed. θ's minimality implies that for any substitution ρ with a similar chain approximating $\rho\mathbb{C}[\vec{x}]$ the first element not in $\eta(D)$ has index at least k.

  For this contradiction the condition $r_k \notin \eta(D)$ is successively transformed according to the expansion rules. We annotate the immediate successors of a node in $\mathcal{T}_T$ with conditions which, given applicability of the rule, are implied by the conditions at the node. With this implication chain we arrive at a condition $r'_k \notin \eta(D)$ for a smaller r'_k.

At the root we have only one constraint, $\mathbb{C}[\vec{x}] \in D$, and the condition $r_k \notin \eta(D)$, but constructor decomposition might introduce nodes with more than one constraint. In this case we will have multiple condition parts. For a set of constraints $\{t_1 \in D_1, \dots, t_n \in D_n\}$ we will have the condition $r^1 \notin \eta(D_1) \vee \dots \vee r^n \notin \eta(D_n)$. Furthermore a direct successor will satisfy $r^m \leq_c \theta\vec{t}^m$ if that was satisfied at its predecessor. Quite obviously, we could use additional indices to differentiate the conditions from the transformed conditions as well as to relate the transformed conditions to the conditions at the root, but we feel that this would not help but rather confuse the reader.

Before presenting the transformations associated with the expansion rules we narrow down the rules possible on the narrow path, $\mathcal{Q}$ from $\mathbb{C}[\vec{x}] \in D$ at the root to $\mathbb{C}[\vec{t}] \in D$ at the leaf. $\mathcal{Q}$ may not contain some of the rules:

(reuse): This rule is excluded as a precondition for applying rule (loopdecomp).

closing rules: $\mathcal{Q}$ would end too early and would not reach the leaf.

rules removing constraints: Again, there would be no such path.

$\mathcal{Q}$ will consist of the rules (casep), (union), (red), (absred) and (decomp).

We consider each of these rules in turn.

(red): We have a node with the constraints $\{t_1 \in D_1, \dots, t_n \in D_n\}$ and $t_m \to_{\text{no}} t'_m$. We do not change the conditions for the successors. The implication from the node to its successor is trivially satisfied. By theorem 2.93 we conclude $r^m \leq_c \theta t_m \implies r^m \leq_c t'_m$.

(union): The mth constraint is $t_m \in \langle D^1_m, \dots, D^{n_m}_m \rangle$. From $r^m \notin \eta(D_m)$ we conclude $\bigwedge_i r^m \notin \eta(D^i_m)$. Thus we obtain the condition that for every successor its condition part must be satisfied. We change the mth part of the annotation to $r^m \notin \eta(D^i_m)$ for successor i.

(decomp): The mth constraint is $t_m \in D_m$ where $t_m \equiv \mathtt{c}\ t^1_m \dots t^{\alpha(\mathtt{c})}_m$ and $D_m \equiv \mathtt{c}\ D^1_m \dots D^{\alpha(\mathtt{c})}_m$. Since $r^m \leq_c t_m$ there are two cases: $r^m \equiv \mathtt{bot}$ and $r^m \equiv \mathtt{c}\ r^{m,1} \dots r^{m,\alpha(\mathtt{c})}$. In the first case $r^m \notin \eta(D_m)$. In the second case $r^m \notin \eta(D_m)$ implies that at least for one argument $r^{m,i} \notin \eta(D^i_m)$ needs to hold. So in this case we replace the mth condition part with $r^{m,1} \notin \eta(D^1_m) \vee \dots \vee r^{m,\alpha(\mathtt{c})} \notin \eta(D^{\alpha(\mathtt{c})}_m)$. Since r^m and t_m start with the same constructor, we use lemma 2.95 and obtain $\forall i : r^{m,i} \leq_c \theta t^i_m$.

(absred): The mth constraint is $\mathbb{R}[\mathtt{bot}] \in D_m$ and since $\theta\mathbb{R}[\mathtt{bot}] \equiv_c \theta\mathtt{bot} \equiv_c \mathtt{bot}$ the only possibility for r^m is $r^m \equiv \mathtt{bot}$ and we keep the condition $r^m \notin \eta(D_m)$.

(casep): The mth constraint is $t_m \in D_m$ with $\theta t_m \equiv \mathtt{case}_A\ (\mathtt{c}_{A,i}\ t^1_m \dots t^{\alpha(\mathtt{c})}_m)\ e_1 \dots e_{|A|}$. The condition is passed to the successors without change.

Next we need to show that r_k must exhibit enough structure to allow for the transformations along $\mathcal{Q}$ to be applied. We show that for r_l which are too shallow $r_l \in \eta(D)$ must hold. $r_i \not\equiv_c r_j$, if $i \neq j$ is a consequence of $r_i \leq_c r_{i+1}$ and r_{i+1} being exactly one level deeper than r_i and being formed from

constructors and $\mathtt{bot}$ only. It follows that $\exists \mathbb{C}[\cdot], \mathtt{c}_{A,j} : r_i \equiv \mathbb{C}[\mathtt{bot}] \wedge r_{i+1} \equiv \mathbb{C}[\mathtt{c}_{A,j}\ r^1_{i+1} \dots r^{\alpha(\mathtt{c}_{A,j})}_{i+1}]$. If we assume we would have to leave $\mathcal{Q}$ with r_l or that we could only follow it partially, but not to $\mathbb{C}[\vec{t}] \in D$, then there is a first union decomposition along $\mathcal{Q}$, for which r^m has been reduced to $\mathtt{bot}$ and for which, due to the immediate union representation, only the successor with demand $\mathtt{c}_{A,j}\ D^1_m \dots D^{\alpha(\mathtt{c}_{A,j})}_m$ may lie on $\mathcal{Q}$. Since the union D_m is $\mathtt{bot}$-closed we can assume a successor of the union decomposition having $t_m \in \mathtt{Bot}$ as its constraint. For this successor then, $r^m \in \eta(\mathtt{Bot})$ holds and thus $r_l \in \eta(D)$ holds as well. We conclude that we can transform the condition $r_k \notin \eta(D)$ according to the decompositions along $\mathcal{Q}$. For the (casep), the (absred) and the (red) rules there is no transformation of the condition and for the union decomposition we only transform the demand, but not the expression. It follows that r_k has enough structure to allow for all the transformations along $\mathcal{Q}$, and that a condition $r'_k \notin \eta(D)$ is derived for the constraint $\mathbb{C}[\vec{t}] \in D$. r'_k has depth strictly smaller than r_k, because at least one (decomp) rule is on $\mathcal{Q}$. This contradicts the choice of θ.

It remains to show that the condition parts corresponding to the VCs $\{x_1 \in D_1, \dots, x_n \in D_n\}$ cannot be satisfied either.

Along the narrow paths $\mathcal{P}_1, \dots, \mathcal{P}_n$ from the root to $\{x_1 \in D_1, \dots, x_n \in D_n\}$ respectively, applications of the (reuse) rule are also allowed. An argument as for $\mathcal{P}$ shows that r_j must exhibit enough structure to allow for the decompositions along the $\mathcal{P}_i$. If the (reuse) rule is applied on a path one can argue accordingly for the $r^{\vec{m}}$ since the demand N introduced by the (reuse) rule is $\mathtt{bot}$-closed according to lemma 4.55. At the leaf $r^i \leq_c \theta x_i$ holds, but since

$\theta = \theta'\sigma_R\sigma$, and so $\theta x_i \equiv \theta'\sigma_R x_i \in \eta(D_i)$, since the r^i are primitive, and since the D_i are `bot`-closed below the top constructor (or even `bot`-closed if $\texttt{bot} \in \gamma(D_i)$) it follows that $r^i \in \eta(D_i)$.

Thus there cannot be such an $r_k \notin \eta(D)$, so $\theta\mathbb{C}[\vec{x}] \in \gamma(D)$ i.e. θ solves the root. □

Theorem 4.63. *CADE is externally complete.*

Proof. To be shown: if θ is a c-solution for the root, $\mathbb{C}[\vec{x}] \in D$, then it is contained in the semantics of the standard representation, $\mathcal{T}_N$, i.e.

$$\theta\mathbb{C}[\vec{x}] \in \gamma(D) \cap WT_{\Downarrow} \implies \theta\vec{x} \in \gamma(N) \wedge \theta\mathbb{C}[\vec{x}] \in WT_{\Downarrow}.$$

We will show for a primitive substitution θ, that

$$\theta\mathbb{C}[\vec{x}] \in \gamma(D) \cap WT_{\Downarrow} \implies \theta\vec{x} \in \eta(N).$$

With theorem 3.94 we can then deduce for non-primitive θ that

$$\theta\mathbb{C}[\vec{x}] \in \gamma(D) \cap WT_{\Downarrow} \implies \theta\vec{x} \in \gamma(N).$$

Assume there is a counter-example, i.e. there exists a smallest primitive ground substitution $\theta : \theta\mathbb{C}[\vec{x}] \in \gamma(D) \cap WT_{\Downarrow}$, but $\theta\vec{x} \notin \eta(N)$ for $\{x_1, \dots, x_n\} = \mathcal{T}_{\mathcal{V}}$.
There may be some paths, $\mathcal{P}_i$, leading to leaves L_i with nodes N_i directly above L_i, such that for primitive c-solutions θ_i of N_i : $\theta_i\sigma_i = \theta$ where σ_i is the substitution along $\mathcal{P}_i$. Due to the local completeness of the rules there is at least one such path. There may be more than one, because branches may be introduced by union decomposition along such paths.
Next we treat the different cases for closing path $\mathcal{P}_i$ arriving at a contradiction to our assumption in every one of them.

- L_i is solved. L_i contributes a component $\top\dot{\sigma}_R\dot{\sigma}_i\vec{x}$ to the standard representation. Since θ_i is a primitive c-solution of N_i, in particular $\theta_i x_j \in \eta(D_j)$ will hold for all $(x_j \in D_j) \in R$. Thus $\theta_i\sigma_i\vec{x}$ is in the semantics of the standard representation in contradiction to the assumption.

- $\mathcal{P}_i$ is closed with the (c-loop) rule. Hence $N_i = \{\mathbb{C}[\vec{t^1}] \in D, \dots, \mathbb{C}[\vec{t^m}] \in D\} \cup R$ with $\mathit{size}(\sigma_i\vec{x}) > \mathit{size}(\vec{t^j})$ and therefore $\mathit{size}(\theta_i\sigma_i\vec{x}) = \mathit{size}(\theta\vec{x}) > \mathit{size}(\theta_i\vec{t^j})$. We define $\theta^j\vec{x} \overset{\text{def}}{=} \theta_i\vec{t^j}$. $\vec{t^j}$ consists of constructors, variables and `bot` and so θ^j has smaller primitive substitutes than θ. From lemma 4.54 we know that $\theta_i\mathbb{C}[\vec{t^j}] \equiv \theta_i\vec{t^j}_{\mathcal{T}_{\mathcal{V}}}\mathbb{C}[\vec{x}] \in WT_{\Downarrow}$ and thus θ^j is a c-solution for the root.

  By our choice of θ it must be that $\theta^j\vec{x} \in \eta(N)$ and so $\forall j : \exists k_j : \theta^j\vec{x} \in \eta_{\Delta^{k_j}(\mathbf{0})}(N)$. Let k be the maximum of the k_j. Then $\forall j : \theta_i\vec{t^j} \in \eta_{\Delta^k(\mathbf{0})}(N)$. Since $\theta_i\vec{t^j}$ is primitive and since $\vec{t^j}$ consists of constructors, variables and `bot` only, there is a ρ with $\forall j : \rho \in \Sigma(\dot{\vec{t^j}}) \wedge \theta_i\vec{t^j} \in \Lambda_p(\rho\dot{\vec{t^j}})$. Using the fact that θ_i is a c-solution of R, because it c-solves N_i, and that the free variables in the $\vec{t^i}$ and the variables in R are disjoint, we conclude

$$\begin{aligned}\theta_i\sigma_i\vec{x} &= \theta_i'\sigma_{R,i}\sigma_i\vec{x} \\ &\in \bigcup_{\rho\in\Sigma(\dot{\vec{t^j}}):\forall j:\rho\dot{\vec{t^j}}\sqsubseteq_p^{k+1}N} \eta_{\Delta^{k+1}(\mathbf{0})}(\rho\dot{\sigma}_{R,i}\dot{\sigma}_i\vec{x}) \\ &= \eta_{\Delta^{k+1}(\mathbf{0})}(\top\dot{\sigma}_{R,i}\dot{\sigma}_i\vec{x}\ \texttt{where}\ [\dot{\vec{t^1}}, \dots, \dot{\vec{t^m}}] = [N, \dots, N])\end{aligned}$$

  So in this case we also find a contradiction to the assumption. Recall that $\top$ substitutes only those demand variables that remain unbound by the `where`-expression.

· $\mathcal{P}_i$ was closed with the rule (noloop). Hence there is a normal-order reduction on the narrow path to $\mathbb{C}[\vec{t^j}] \in D$, but no constructor decomposition. Without loss of generality, the node N_i has constraint set $\{\mathbb{C}[\vec{t}] \in D\} \cup R$, since $\sigma_i\vec{x} \equiv \vec{t^j}$ for all j. We know that θ c-solves the root and that $\theta\mathbb{C}[\vec{x}] \equiv \theta_i\vec{t_{\vec{x}}}\mathbb{C}[\vec{x}] \equiv \theta_i\mathbb{C}[\vec{t}]$. Then, obviously, $\theta_i\mathbb{C}[\vec{t}] \in WT_{\Downarrow}$.

Among the rules which may appear on the narrow path, only (decomp), (reuse), (absred) and (red) can change the expression, but we do not allow (reuse), and (decomp) is excluded by assumption, so the only change the expression may have experienced is by (red) or (absred). Then $\sigma\mathbb{C}[\vec{x}] \xrightarrow{+}_{\text{no}} \mathbb{C}[\vec{t}] \equiv \sigma\mathbb{C}[\vec{x}]$. Since normal-order reduction is possible at all, $\sigma\mathbb{C}[\vec{x}]$ cannot be a WHNF. But then $\sigma\mathbb{C}[\vec{x}]\Uparrow$ must hold, and there cannot be a θ_i satisfying $\theta_i\sigma_i\mathbb{C}[\vec{x}] \in \eta(D)$, since $\texttt{Bot} \not\leq_\gamma D$. Again we found a contradiction to the choice of θ.

· In the remaining two cases, i.e. if $\mathcal{P}$ was closed either with the rule (loopdecomp) or with the rule (loopred), θ_i solves N_i and with lemma 4.34 we deduce that θ_i solves L_i as well. An argument similar to the one for solved leaves shows that $\theta\vec{x} \in \eta(N)$ in contradiction to the assumption. □

5 Extensions and Implementation

While the base calculi have been theoretically investigated and in particular have been proved sound and complete, this chapter introduces extensions of the calculi and discusses some of the implementation's aspects. The emphasis in this chapter is on variety rather than on full integration of the extensions with the base calculi.

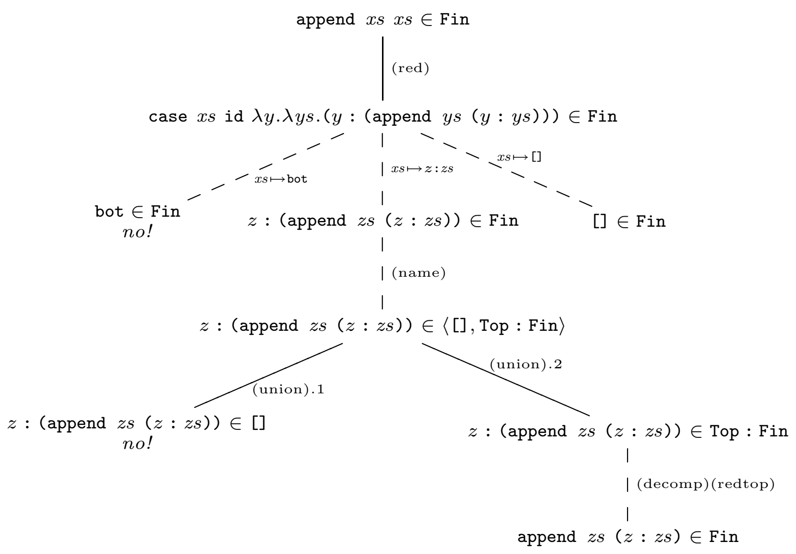
5.1 Multiply occurring variables

We have restricted the calculi to inputs in which each root variable occurs exactly once. In this section we will motivate the decision and show how analysis results for constraints with multiply occurring root variables can be computed from results for similar constraints using unique variables. We show that the result thus computed is externally sound and complete whenever the result used for the computation is externally sound and complete, respectively.

5.1.1 Problem

It is not possible to directly solve every input constraint in which a variable occurs more than once with our calculi.

Example 5.1. An example will illustrate this. The constraint `append` $xs\ xs \in$ `Fin` can lead to the tableau in figure 5.1.



append xs xs ∈ Fin
(red)
case xs id λy.λys.(y : (append ys (y : ys))) ∈ Fin
xs↦bot
xs↦z:zs
xs↦[]
bot ∈ Fin
no!
z : (append zs (z : zs)) ∈ Fin
[] ∈ Fin
(name)
z : (append zs (z : zs)) ∈ ⟨[], Top : Fin⟩
(union).1
(union).2
z : (append zs (z : zs)) ∈ []
no!
z : (append zs (z : zs)) ∈ Top : Fin
(decomp)(redtop)
append zs (z : zs) ∈ Fin

Figure 5.1: Open tableau with multiply occurring variables

It is easily seen, that this tree could grow infinitely, at a node `append` $zs_i\ (z : z_1 : \ldots : z_i : zs_i) \in$ `Fin` assuming $zs_i \mapsto z_{i+1} : zs_{i+1}$ and arriving after application of (name), (union).2, (decomp), (redtop) at the constraint `append` $zs_{i+1}\ (z : z_1 : \ldots : z_{i+1} : zs_{i+1}) \in$ `Fin`. No global rule can be applied since there is no substitution σ mapping σxs to zs_{i+1} and σxs to $z : z_1 : \ldots : z_{i+1} : zs_{i+1}$.

5.1.2 Solution

If on the other hand we have analysis results for an input constraint using only distinct variables, we can construct solutions for all similar constraints that differ only by having multiple occurrences of variables from the former. To construct the solution for the latter we simply intersect the components in the former where the latter names the same variable.

Example 5.2. We can analyze `append` $xs\ ys \in$ `Fin` and obtain the result `append` $xs\ ys \in$ `Fin` $\rightarrow_{\text{CADE}} N$ for a standard representation $N = \langle$`[[],Fin]`$, [$`Top :` $zs, ys]$ `where` $[zs, ys] = N\rangle$. Using demand transformations it is easy to see that $N \equiv_\gamma$ `[Fin,Fin]`. Hence a solution for `append` $xs\ xs \in$ `Fin` is $xs \cap ys$ `where` $[xs, ys] = N$, which is $\equiv_\gamma$-equivalent to `Fin`.

This construction only depends on the root constraints and the standard representation and is completely independent of the expansion rules and the tree representation. Thus it is no restriction to limit the calculi to inputs in which the variables are distinct.
We generalize the example and then prove that the construction leads to an externally sound tableau if the tableau it is based on is externally sound. Let $\mathcal{T}$ be a closed tableau for root $R = \{r_1 \in C_1, \ldots, r_k \in C_k\}$ and root variables $\mathcal{T}_\mathcal{V} = (x_1, \ldots, x_n)$ (each occurring exactly once in R) and let $\mathcal{T}_N$ be its standard representation: $N = \langle e_1, \ldots, e_s\rangle$. Let R' be the root of the tree $\mathcal{T}'_T$ and let $R' = \{r'_1 \in C_1, \ldots, r'_k \in C_k\}$ be obtained from R by consistently renaming some variables in the r_i. I.e. there is a substitution $\sigma : \mathcal{T}_\mathcal{V} \rightarrow \mathcal{T}_\mathcal{V}$ with $R' \equiv \sigma R$ and we can partition the root variables into sets $I_{i_1}, \ldots, I_{i_m} \subseteq \mathcal{T}_\mathcal{V}$ where the i_j differ pairwise such that $\sigma = \{x \mapsto x_{i_j} | 1 \leq j \leq m \wedge x \in I_{i_j}\}$. We can regard the following as the standard representation $\mathcal{T}'_{N'}$ for $\mathcal{T}'$:

$$N' = [\bigcap_{i \in I_{i_1}} x_i, \ldots, \bigcap_{i \in I_{i_m}} x_i] \ \texttt{where} \ [x_1, \ldots, x_n] = N.$$

Theorem 5.3. *The construction for multiply occurring variables obtains an externally sound tableau from such a tableau.*

Proof. We need to show that $\theta[x_{i_1}, \ldots, x_{i_m}] \equiv \theta\mathcal{T}'_\mathcal{V} \in \eta(N')$ implies $\theta r'_i \in \gamma(C_i)$ for all i provided that $\theta\mathcal{T}_\mathcal{V} \in \eta(N) \implies \forall i : \theta r_i \in \gamma(C_i)$.

$$\begin{aligned}
& \theta\mathcal{T}'_\mathcal{V} \in \eta_{\Delta^i(\mathbf{0})}(N') \\
\Longrightarrow\quad & \\
& \theta\mathcal{T}'_\mathcal{V} \in \bigcup_{\dot\rho \in \Sigma(\mathcal{T}_\mathcal{V}) : \dot\rho\mathcal{T}_\mathcal{V} \sqsubseteq^i_p N} \eta_{\Delta^i(\mathbf{0})}(\dot\rho[\bigcap_{i \in I_{i_1}} x_i, \ldots, \bigcap_{i \in I_{i_m}} x_i]) \\
\Longrightarrow\quad & \\
& \exists\dot\rho \in \Sigma(\mathcal{T}_\mathcal{V}) \forall j : \theta x_{i_j} \in \eta_{\Delta^i(\mathbf{0})}(\bigcap_{i \in I_{i_1}} \dot\rho x_i) \wedge \dot\rho\mathcal{T}_\mathcal{V} \sqsubseteq^i_p N
\end{aligned}$$

In particular such a ρ is a primitive solution of R, thus $\forall i : \rho r_i \in \gamma(C_i)$ and since r_i and r'_i differ only by identification of some variable names we conclude $\theta r'_i \in \gamma(C_i)$. □

In general, $\gamma(C) \cap \gamma(D) \not\subseteq \gamma(C \cap D)$, and thus external completeness will require additional conditions.

Example 5.4. Assume `InfEven` $=$ $\langle$`Bot,Top : Top : InfEven`$\rangle$ and `InfOdd` $=$ $\langle$`Top : Bot,Top : Top : InfOdd`$\rangle$. A tableau $\mathcal{T}$ for input `c` $x\ y \in$ `c InfEven InfOdd` can be closed obtaining $N = \langle$`InfEven,InfOdd`$\rangle$ as its standard representation $\mathcal{T}_N$. With the above construction we obtain $N' = x \cap y$ `where` $[x, y] = N$ with $N' \equiv_\gamma \emptyset$ as the standard representation of `c` $x\ x \in$ `c InfEven InfOdd`. However, $\theta = \{x \mapsto$ `repeat 1`$\}$ c-solves this root and thus the construction is not complete.

This incompleteness is much less a problem than it seems, because often $\gamma(C) \cap \gamma(D) \subseteq \gamma(C \cap D)$. It should suffice if all but one of the demands in an intersection are either finite or `bot`-closed.

Conjecture 5.5. *Let $D_1, \dots, D_n \in \Lambda_C$. If for at most one $i : D_i$ is neither `bot`-closed nor $\eta(D_i) \neq \gamma(D_i)$ then*

$$\gamma(D_1) \cap \dots \cap \gamma(D_n) = \gamma(D_1 \cap \dots \cap D_n).$$

The criterion which the demands will have to meet for this construction to be complete will not be provided here, and consequently, neither will the statement of the construction's completeness. We conjecture that it suffices if for any of the $i_j : \gamma(\bigcap_{i \in I_{i_j}} x_i$ `where` $[x_1, \dots, x_n] = N) = \bigcap_{i \in I_{i_j}} \gamma(x_i$ `where` $[x_1, \dots, x_n] = N)$. (Intersection taking precedence over `where`-binding.)

5.2 Local rules

5.2.1 Case

We can apply the (casep) rule if the first argument of the $\texttt{case}_A$-expression is a variable. A generalization of this rule allows application of this strategy to $\texttt{case}_A$-expressions in which the first argument is not a WHNF. We could try the following generalization:
Let $M_i = \{\mathbb{R}[\texttt{case}_A\ (\texttt{c}_{A,i}\ x_{i,1} \dots x_{i,\alpha(\texttt{c}_{A,i})})\ e_1 \dots e_{|A|}] \in C, t \in \texttt{c}_{A,i}\ \texttt{Top} \dots \texttt{Top}\} \cup R$ and $M_0 = \{\mathbb{R}[\texttt{case}_A\ \texttt{bot}\ e_1 \dots e_{|A|}] \in C, t \in \texttt{Bot}\} \cup R$

$$\frac{\{\mathbb{R}[\texttt{case}_A\ t\ e_1 \dots e_{|A|}] \in C\} \cup R}{M_0 \mid M_1 \mid \dots \mid M_{|A|}} \qquad \text{(caseexp1)}$$

But (caseexp1) is not sound.

Example 5.6. Let $\texttt{f} \overset{\text{def}}{=} \texttt{1 : []}$ and let $C \equiv \texttt{2 : []}$, then $\texttt{case}_{\texttt{List}}\ \texttt{f}\ x\ \texttt{(:)} \xrightarrow{*}_{\text{no}} \texttt{1 : []}$ so there cannot be a solution for $\texttt{case}_{\texttt{List}}\ \texttt{f}\ x\ \texttt{(:)} \in \texttt{2 : []}$. But as we see in figure 5.2 the calculi would find solutions.

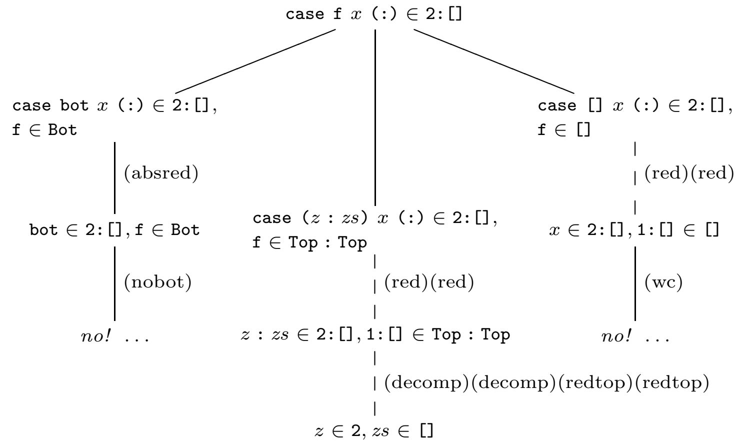

case f x (:) ∈ 2:[]
case bot x (:) ∈ 2:[], f ∈ Bot
(absred)
bot ∈ 2:[], f ∈ Bot
(nobot)
no! ...
case (z : zs) x (:) ∈ 2:[], f ∈ Top : Top
(red)(red)
z : zs ∈ 2:[], 1:[] ∈ Top : Top
(decomp)(decomp)(redtop)(redtop)
z ∈ 2, zs ∈ []
case [] x (:) ∈ 2:[], f ∈ []
(red)(red)
x ∈ 2:[], 1:[] ∈ []
(wc)
no! ...

Figure 5.2: Unsound tableau using (caseexp1)

The analysis would thus erroneously produce `Top` as the result for this input. In order to solve this problem we could e.g. use the (reuse) rule to perform the analysis in two steps. In the first step we analyse $\mathbb{R}[\texttt{case}_A\ x_0\ e_1 \dots e_{|A|}] \in C$ for a new variable x_0 to obtain a result N. In the second step we apply the (reuse) with the result from the first, so if $x_0, x_1, \dots, x_n$ are the free variables in $\mathbb{R}[\texttt{case}_A\ x_0\ e_1 \dots e_{|A|}]$ we add a successor $[t, x_1, \dots, x_n] \in N, R$. This approach will cover many cases, but has the disadvantage that completeness is lost due to the incompleteness of (reuse) and some expansion rules are excluded for the remaining analysis if global rules are to be applied.

Alternatively, we could also use the fresh variables $\dot{x}_{i,1}, \ldots, \dot{x}_{i,\alpha(\mathtt{c}_{A,i})}$ as demand variables in the demand constraining t instead of the `Tops`. As a consequence the rule (casep) needs to appropriately substitute the demand as well as the Λ-expression in a constraint. Additionally, the proof of external soundness needs to be modified. Furthermore, the (reuse) rule will need modification to reuse results for analyses involving demand variables. The case considering a leaf closed with the (loopred) rule will need to provide for this (caseexp) rule. This latter approach is the one we have taken for the implementation, thus the (caseexp) rule in the implementation is:

Let $M_i = \{\mathbb{R}[\mathtt{case}_A\ (\mathtt{c}_{A,i}\ x_{i,1} \ldots x_{i,\alpha(\mathtt{c}_{A,i})})\ e_1 \ldots e_{|A|}] \in C, t \in \mathtt{c}_{A,i}\ \dot{x}_{i,1} \ldots \dot{x}_{i,\alpha(\mathtt{c}_{A,i})}\} \cup R$ and $M_0 = \{\mathbb{R}[\mathtt{case}_A\ \mathtt{bot}\ e_1 \ldots e_{|A|}] \in C, t \in \mathtt{Bot}\} \cup R$

$$\frac{\{\mathbb{R}[\mathtt{case}_A\ t\ e_1 \ldots e_{|A|}] \in C\} \cup R}{M_0 \mid M_1 \mid \ldots \mid M_{|A|}} \qquad \text{(caseexp)}$$

5.2.2 Redundant `where`

There are circumstances under which we can simplify constraints using `where`-expressions to not use `where`-expressions. This is desirable because constraints without `where`-expressions are usually much more easily employed in other analyses and are more comprehensible for the programmer.

If a contribution consists only of variables from the pattern of a `where`-expression then we can replace that `where`-expression by substituting the expressions into the pattern.

Let $\dot{T}$ be a demand pattern and let $\mathcal{FV}(T) = \{v_1, \ldots, v_n\}$

$$\frac{\{[t_1, \ldots, t_n] \in [v_1, \ldots, v_n]\ \mathtt{where}\ \dot{T} = N\} \cup R}{\{\{v_1 \mapsto t_1, \ldots, v_n \mapsto t_n\}T \in N\} \cup R} \qquad \text{(redwhere)}$$

Theorem 5.7. *The* (redwhere) *rule is sound and complete.*

Proof. Let $\tau = \{v_1 \mapsto t_1, \ldots, v_n \mapsto t_n\}$.

sound: Let $\theta\tau T \in \gamma(N)$, then there is an ascending chain $s_1 \leq_c s_2 \leq_c \ldots$ in $\eta(N)$ with $\theta\tau T \equiv_c \bigsqcup_i^c s_i$. Since T consists of constructors, variables and `bot` only, by lemma 3.91 there is some i_0 such that for all $i > i_0 : \exists\sigma_i : s_i \equiv_c \sigma_i T$. The substitutes of the σ_i are all primitive and so by lemma 3.25 there is a primitive demand for every one of them. We can convert each σ_i to a substitution $\dot{\sigma}_i$ which substitutes primitive demands only and for which $s_i \in \Lambda_p(\dot{\sigma}_i\dot{T})$ for sufficiently large i. All the s_i are in $\eta(N)$ and with lemma 3.26 we conclude $\dot{\sigma}_i\dot{T} \sqsubseteq_p N$. So for all but finitely many of the $s_i : \sigma_i[v_1, \ldots, v_n] \in \bigcup_{\sigma\in\Sigma(\dot{T}):\sigma\dot{T}\sqsubseteq_p N} \eta(\sigma[v_1, \ldots, v_n])$. From continuity of contexts we conclude $\bigsqcup_i^c \sigma_i[v_1, \ldots, v_n] \equiv_c \theta\tau[v_1, \ldots, v_n]$ and hence $\theta[t_1, \ldots, t_n] \in \gamma([v_1, \ldots, v_n]\ \mathtt{where}\ \dot{T} = N)$.

The constrained variables of the node and its predecessor M are identical making θ a solution of M.

complete: Let $\theta[t_1, \ldots, t_n] \in \gamma([v_1, \ldots, v_n]\ \mathtt{where}\ \dot{T} = N)$, then there is an ascending chain $s_1 \leq_c s_2 \leq_c \ldots$ in $\eta([v_1, \ldots, v_n]\ \mathtt{where}\ \dot{T} = N)$ satisfying $\bigsqcup_i^c s_i \equiv_c \theta[t_1, \ldots, t_n] \equiv \theta\tau[v_1, \ldots, v_n]$. By lemma 3.38 $\eta([v_1, \ldots, v_n]\ \mathtt{where}\ \dot{T} = N) = \bigcup_{\sigma\in\Sigma(\dot{T}):\sigma\dot{T}\sqsubseteq_p N} \eta(\sigma[v_1, \ldots, v_n])$ and for every i there is some $\dot\sigma_i : \dot\sigma_i \dot T \sqsubseteq_p N \wedge s_i \in \eta(\dot\sigma_i[v_1, \ldots, v_n])$. Furthermore, we obtain $\bigsqcup_i^c \sigma_i[v_1, \ldots, v_n] \equiv_c \theta\tau[v_1, \ldots, v_n]$ and thus $\theta\tau T \in \gamma(N)$. We have proved the more general case not requiring θ to satisfy the $WT_\Downarrow$ condition at the root. This implies the restricted case where θ meets this condition. □

5.2.3 Focus

We can lift those components from under a `where`-expression which do not have any free variables since the appropriate demands cannot be affected by any substitution.
Let $\mathcal{FV}(D_1) = \emptyset$

$$\frac{\{[s_1, \ldots, s_n] \in [D_1, \ldots, D_n] \texttt{ where } T = N\} \cup R}{\{s_1 \in D_1, [s_2, \ldots, s_n] \in [D_2, \ldots, D_n] \texttt{ where } T = N\} \cup R} \quad \text{(focus)}$$

Theorem 5.8. *The rule* (focus) *is sound and complete.*

Proof.

sound: Let θ be a solution of the successor, so $\theta s_1 \in \gamma(D_1) \wedge \theta[s_2, \ldots, s_n] \in \gamma([D_2, \ldots, D_n] \texttt{ where } T = N)$.

There are ascending chains $r_1 \leq_c r_2 \leq_c \ldots$ in $\eta(D_1)$ as well as $t_1 \leq_c t_2 \leq_c \ldots$ in $\eta([D_2, \ldots, D_n] \texttt{ where } T = N)$ such that $\bigsqcup_i^c r_i \equiv_c \theta s_1$ and $\bigsqcup_i^c t_i \equiv_c \theta[s_2, \ldots, s_n]$. The t_i each have an SCWHNF $[t_i^2, \ldots, t_i^n]$. From continuity of clubs and theorem 2.125 it follows that $[\bigsqcup_i^c r_i, \bigsqcup_i^c t_i^2, \ldots, \bigsqcup_i^c t_i^n] \equiv_c \bigsqcup_i^c [r_i, t_i^2, \ldots, t_i^n] \equiv_c \theta[s_1, \ldots, s_n]$.

Since $\eta([D_2, \ldots, D_n] \texttt{ where } T = N) = \bigcup_{\sigma \in \Sigma(T) : \sigma T \sqsubseteq_p N} \eta(\sigma[D_2, \ldots, D_n])$ and $\mathcal{FV}(D_1) = \emptyset$ we know that $\forall i : [r_i, t_i^2, \ldots, t_i^n] \in \eta([D_1, \ldots, D_n] \texttt{ where } T = N)$, and

since the same variables occur free in the predecessor and in the successor, θ will be a solution of the predecessor.

complete: The reasoning is quite similar to the above. Again the reasoning succeeds in the general case without resorting to the $WT_\Downarrow$ condition and implies the more restricted case. □

Example 5.9. Suggesting applications of the (focus) rule are e.g. lifting `Top`, `Fun` or `Bot` out from the contribution of a `where`-expression. In this way a tableau with a leaf $[\texttt{bot}, s, t] \in [\texttt{Bot}, \texttt{Top}, x] \texttt{ where } x = \texttt{Inf}$ can be expanded with $\texttt{bot} \in \texttt{Bot}, s \in \texttt{Top}, t \in x \texttt{ where } x = \texttt{Inf}$ and by application of other rules (cf. section 3.7) with $t \in \texttt{Inf}$.

5.2.4 Local rules below `where`

The local rules from the base calculi can be applied below a `where`-expression. We obtain the following rules:

$$\frac{\{[s_1, \ldots, s_n] \in [D_1, \ldots, D_n] \texttt{ where } T = N\} \cup R}{\{[t_1, s_2, \ldots, s_n] \in [D_1, \ldots, D_n] \texttt{ where } T = N\} \cup R} \text{if } s_1 \to_{\text{no}} t_1 \quad \text{(redw)}$$

$$\frac{\{[\texttt{c } t_1 \ldots t_{\alpha(\texttt{c})}, \ldots] \in [\texttt{c } C_1 \ldots C_{\alpha(\texttt{c})}, \ldots] \texttt{ where } T = N\} \cup R}{\{[t_1, \ldots, t_{\alpha(\texttt{c})}, \ldots] \in [C_1, \ldots, C_{\alpha(\texttt{c})}, \ldots] \texttt{ where } T = N\} \cup R} \quad \text{(decompw)}$$

If there is no substitution σ such that $\gamma(\sigma D_1)$ and $\gamma(\texttt{c Top} \ldots \texttt{Top})$ have elements in common, we can apply the rule:

$$\frac{\{[\mathtt{c}\ldots,\ldots]\in[D_1,\ldots,D_n]\ \mathtt{where}\ T=N\}\cup R}{\textit{no!}\ \{[\mathtt{c}\ldots,\ldots]\in[D_1,\ldots,D_n]\ \mathtt{where}\ T=N\}\cup R}\quad\text{(wcw)}$$

$$\frac{\{[\mathbb{R}[\mathtt{bot}],\ldots]\in[D_1,\ldots,D_n]\ \mathtt{where}\ T=N\}\cup R}{\{[\mathtt{bot},\ldots]\in[D_1,\ldots,D_n]\ \mathtt{where}\ T=N\}\cup R}\quad\text{(absredw)}$$

In the following rule let $E_i \overset{\text{def}}{=} [C_i, D_2, \ldots, D_n]\ \mathtt{where}\ T = N$.

$$\frac{\{[s_1,\ldots,s_n]\in[\langle C_1,\ldots,C_m\rangle,D_2,\ldots,D_n]\ \mathtt{where}\ T=N\}\cup R}{\{[s_1,\ldots,s_n]\in E_1\}\cup R|\ldots|\{[s_1,\ldots,s_n]\in E_m\}\cup R}$$
(unionw)

$$\frac{\{[s_1,s_1,s_3,\ldots,s_n]\in[D_1,\ldots,D_n]\ \mathtt{where}\ T=N\}\cup R}{\{[s_1,s_3,\ldots,s_n]\in[D_1\cap D_2,\ldots,D_n]\ \mathtt{where}\ T=N\}\cup R}\quad\text{(isw)}$$

And if the (casep) rule could be applied to a leaf $\{\mathbb{R}[\mathtt{case}_A\ x\ \vec{e}]\}\cup R$ attaching new leaves with edges labeled $\rho_i, 1 \le i \le |A|$ we define $L_i \overset{\text{def}}{=} \rho_i(\{[\mathbb{R}[\mathtt{case}_A\ x\ \vec{e}], s_2, \ldots, s_n] \in [D_1, \ldots, D_n]\ \mathtt{where}\ T = N\} \cup R)$.

$$\frac{\{[\mathbb{R}[\mathtt{case}_A\ x\ \vec{e}],s_2,\ldots,s_n]\in[D_1,\ldots,D_n]\ \mathtt{where}\ T=N\}\cup R}{L_0|\ldots|L_{|A|}}$$
(casepw)

It would not make sense to define a similar rule for the (reuse) rule, since a demand which could be used for this rule does not have any free variables but then (focus) could be applied. We leave it

to future research to determine a general criterion for local rules to be applicable below `where`-expressions and to prove that any local rule meeting this criterion leads to a sound or complete rule if this original rule was sound or complete, respectively.

5.2.5 Closure

The local rules below `where`-expressions can lead to a joc having only variables as arguments, constrained by a `where`-expression. If this happens we can capture the information encoded in the constraint by an additional label "*loop!*...".

$$\frac{\{[y_1,\ldots,y_m]\in[C_1,\ldots,C_m]\ \mathtt{where}\ T=N\}\cup R}{\textit{loop!}\ \tau\sigma_R\dot{\sigma}[x_1,\ldots,x_n]\ \mathtt{where}\ T=N}\quad\text{(jocclose)}$$

where σ is the substitution along the path from the root to the node with substitutes converted to demands, all the y_i are variables and $\tau = \{\dot{y_i} \mapsto C_i | 1 \le i \le m\}$.

5.3 Higher-order

The treatment of higher-order expressions in the calculi as presented in sections 4.4 and 4.5 is comparatively coarse. For expressions that are saturated constructor WHNFs demands are available precisely differencing each constructor, but all functions or unsaturated CWHNFs are represented in one demand: `Fun`. In this section we will see extensions allowing demands for FWHNFs with a level of detail similar to that of demands for SCWHNFs.
In his master's thesis, Dirk Rehberger [Reh99] elaborated these extensions and their relation to the base calculi. Full theoretical integration of these extensions into the base calculi remains future

work. They will be presented here and we will discuss what is necessary for full integration.
If we have a constraint $f \in C$ and f is an application of a variable to one or more arguments, all the rules from the base calculi can do is either remove the constraint completely or operate on the demand side. No rule of the base calculi will allow us to decompose or reduce f. Recall from definition 2.36 that f may become an FWHNF by appropriate substitution, but it may also become an SCWHNF. We employ the extension to $\overset{\forall}{\rightarrow}$-demands, which we will show to be able to handle this case.

5.3.1 Demands

Up to now the only possibility to represent higher-order Λ-expressions was by the demand `Fun`. We will now introduce demands allowing a more precise analysis of constraints involving higher-order expressions.
The syntax of demand expressions in definition 3.1 is extended by the productions

$$\begin{array}{lll} \text{demand expr} & E' & \rightarrow F \\ \forall - \text{demand} & F & \rightarrow E'_1 \overset{\forall}{\rightarrow} E'_2 \end{array}$$

Example 5.10. This is quite an expressive extension, for which there are some intuitive examples.

1. A function, f, which maps the (data) value $T \in \Lambda$ consisting of constructors, `bot` and FWHNFs into $\gamma(C)$, is in the concretization of the demand $\dot{T} \overset{\forall}{\rightarrow} C$.

2. Demands containing higher-order functions in their concretization can be built as $A \overset{\forall}{\rightarrow} B$ if A or B contain

$\forall$-demands themselves. The former case will, however, be precluded due to the already mentioned reasons: we want to define $\eta^{\rightarrow}(\cdot)$ with a least fixpoint, and if $\Delta^{\rightarrow}(\cdot)$ is monotonous we can be sure of that fixpoint's existence. If its existence could be ensured without requiring $\Delta^{\rightarrow}(\cdot)$ to be monotonous, $\forall$-demands nested in the left of such demands could be reconsidered.

Next we will present a conservative extension, $\eta_\rho^{\rightarrow}$, of the function η_ρ, which will also map the $\forall$-demands to sets of primitive expressions, according to the intuition above.

Definition 5.11 ($\eta_\rho^{\rightarrow}$). The definition of $\eta_\rho^{\rightarrow}$ is identical to that of η_ρ, where every occurrence of η_ρ is replaced by $\eta_\rho^{\rightarrow}$. Additionally, we define

$$\eta_\rho^{\rightarrow}(C_1 \overset{\forall}{\rightarrow} C_2) \overset{\text{def}}{=} \{t \mid \forall u \in \eta(C_1) : (t\ u) \in \eta_\rho^{\rightarrow}(C_2)\} \tag{5.1}$$

We have to use η for C_1 in this definition in order to ensure that our choice for proving the existence of the fixpoint works. If we would allow $\forall$-demands here by using $\eta_\rho^{\rightarrow}(\cdot)$, we would lose monotonicity of $\Delta^{\rightarrow}(\cdot)$ and would consequently have to prove the existence of a fixpoint for a non-monotonous operator.
We state without proof the following lemma.

Lemma 5.12. *Lemma 3.38 remains valid if $\eta(\cdot)$ is replaced by $\eta^{\rightarrow}(\cdot)$.*

5.3.2 Splitting applications

The most simple rule for the higher-order extensions splits an application of a variable to some expressions into two constraints, one for the last argument and one for the higher-order expression resulting from removing that last argument of the application.

$$\frac{\{g\ t \in C\} \cup R}{\{t \in y, g \in y \overset{\forall}{\to} C\} \cup R}, y \text{ is a fresh demand variable} \quad \text{(split)}$$

An intuitive interpretation of the resulting leaf is that t may be constrained by any demand, but whatever that demand may be, g needs to map any element in the concretization of that demand to an element of the concretization of the original demand.

Demand variables

The rule for splitting applications introduces demand variables into the constraints in order to relate the constraints for g and t. It is easy to see that this forces us to allow any node including the root of the tableau to use such demand variables. We will have to modify the definition of solution. This will be a conservative extension, i.e. if no demand variables are present the new and old definitions coincide.

Definition 5.13 (solution in the presence of demand variables). Let $\mathcal{T}$ be a tableau, let θ be a ground substitution and let σ be the (idempotent) substitution along the path to a node $M = \{s_1 \in C_1, \ldots, s_n \in C_n\}$. Furthermore, let $y_1, \ldots, y_r$ be all the demand variables appearing in M. θ is a *solution of the node* M, iff $\theta[s_1, \ldots, s_n] \in \gamma([C_1, \ldots, C_n]\ \texttt{where}\ y_1 = \texttt{Top}; \ldots; y_r = \texttt{Top})$, and $\text{dom}(\theta) \supseteq \mathcal{CV}(M)$.

Binding the demand variables in the `where`-expression serves the purpose of ensuring that each use of a variable will be substituted with the same primitive demand expression.

Remark 5.14. Demand variables are introduced in pairs by the (split) rule. If a demand variable is initially introduced two constraints will use it. The same demand variable will not be introduced into any other constraint, since whenever a demand variable is introduced it will be a fresh one. Lastly, whenever a demand variable is substituted, all its occurrences are substituted.

Theorem 5.15. *The rule* (split) *is internally sound.*

Proof. We need to show that every substitution solving the new leaf also solves the old one. So let θ be a solution of $\{t \in y, g \in y \overset{\forall}{\to} C, s_1 \in C_1, \ldots, s_n \in C_n\}$, then $\theta[t, g, s_1, \ldots, s_n] \in \gamma([y, y \overset{\forall}{\to} C, C_1, \ldots, C_n]\ \texttt{where}\ y = \texttt{Top}; \ldots)$. (Here the ellipses stand for other potentially present variables.)
As in the other soundness proofs, we show soundness with respect to primitive solutions and obtain soundness for the rule by continuity arguments.
So let

$$\begin{aligned}
&\theta[t, g, s_1, \ldots, s_n] \\
&\quad \in \eta^{\to}([y, y \overset{\forall}{\to} C, C_1, \ldots, C_n]\ \texttt{where}\ y = \texttt{Top}; \ldots) \\
\Longrightarrow\ & \text{(lemma 5.12)} \\
&\exists \sigma \in \Sigma_p : \theta[t, g, s_1, \ldots, s_n] \\
&\qquad \in \eta^{\to}(\sigma[y, y \overset{\forall}{\to} C, C_1, \ldots, C_n]\ \texttt{where} \ldots) \\
\Longrightarrow\ & \text{(definition 5.11)} \\
&\exists \sigma \in \Sigma_p : \theta[g\ t, s_1, \ldots, s_n] \\
&\qquad \in \eta^{\to}(\sigma[C, C_1, \ldots, C_n]\ \texttt{where} \ldots) \quad \square
\end{aligned}$$

6 Examples and Applications

This chapter is a collection of various examples and applications. Among the examples are a motivation for a heuristic keeping the tableau size down in an implementation, a demonstration of the advantages of the (reuse) rule, one demonstrating the ability of the base calculi to handle higher-order expressions in specific cases, examples exhibiting the limitations of the calculi, and a larger example analyzing an encoding of the signed Peano numbers with hyperstrict arithmetic operations. The applications include a safety analysis, i.e. a search for an answer to “Can this program crash due to a software defect?”. Some pre- and post-conditions from the calculi of Hoare, Dijkstra and Baber can be expressed as demands and thus for some post-conditions an application of the calculi is finding a pre-condition. Furthermore, absence and need are among the analyses to which the calculi are applied as well as checking the requirements for the safety of an optimization.

6.1 Examples

6.1.1 Reuse

Example 6.1. An analysis which would not lead to a closed tableau without the (reuse) rule is the analysis of `concat` $xs \in$ `Inf`, for which we obtain the tableau in figure 6.1.

In the nodes labeled `foldr (++) []` $zs \in$ `Inf` and

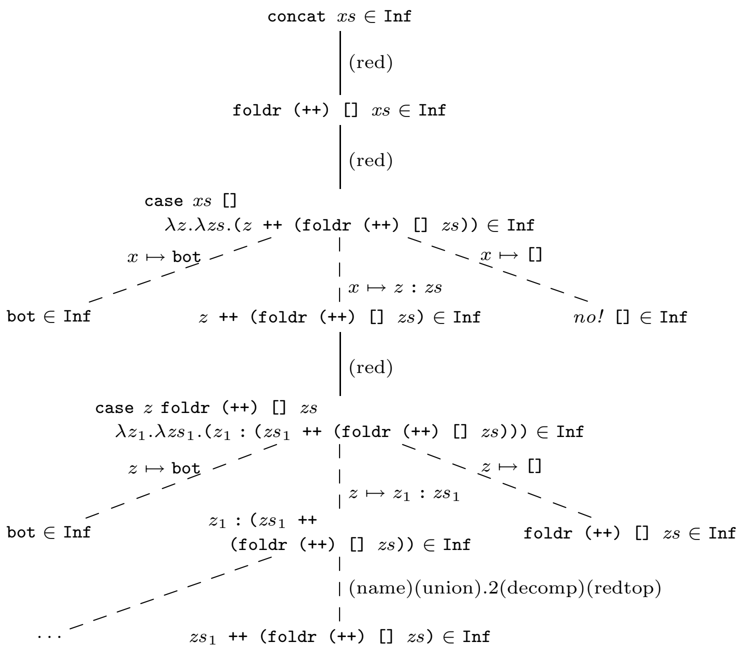

concat $xs \in$ Inf
(red)
foldr (++) [] $xs \in$ Inf
(red)
case xs [] $\lambda z.\lambda zs.$(z ++ (foldr (++) [] zs)) $\in$ Inf
$x \mapsto$ bot
$x \mapsto z : zs$
$x \mapsto$ []
bot $\in$ Inf
z ++ (foldr (++) [] zs) $\in$ Inf
$no!$ [] $\in$ Inf
(red)
case z foldr (++) [] zs $\lambda z_1.\lambda zs_1.(z_1$: (zs_1 ++ (foldr (++) [] zs))) $\in$ Inf
$z \mapsto$ bot
$z \mapsto z_1 : zs_1$
$z \mapsto$ []
bot $\in$ Inf
z_1 : (zs_1 ++ (foldr (++) [] zs)) $\in$ Inf
foldr (++) [] $zs \in$ Inf
(name)(union).2(decomp)(redtop)
...
zs_1 ++ (foldr (++) [] zs) $\in$ Inf

Figure 6.1: Open tableau for `concat` $xs \in$ `Inf`

zs_1 `++ (foldr (++) []` zs`)` $\in$ `Inf` we cannot apply global rules because the top node of the loop would need to be one from below the tableau’s root. The one directly below the root for the former, and the one directly below the first case branch for the latter. If we analyze generalizations of the two constraints separately, i.e. if we analyze `foldr (++) []` $zs \in$ `Inf` and xs `++` $ys \in$ `Inf` we can apply global rules in both cases. We will present the analysis of xs `++` $ys \in$ `Inf` in figure 6.2.

Collecting the result from this tableau, $\mathcal{T}$, we obtain $N =$

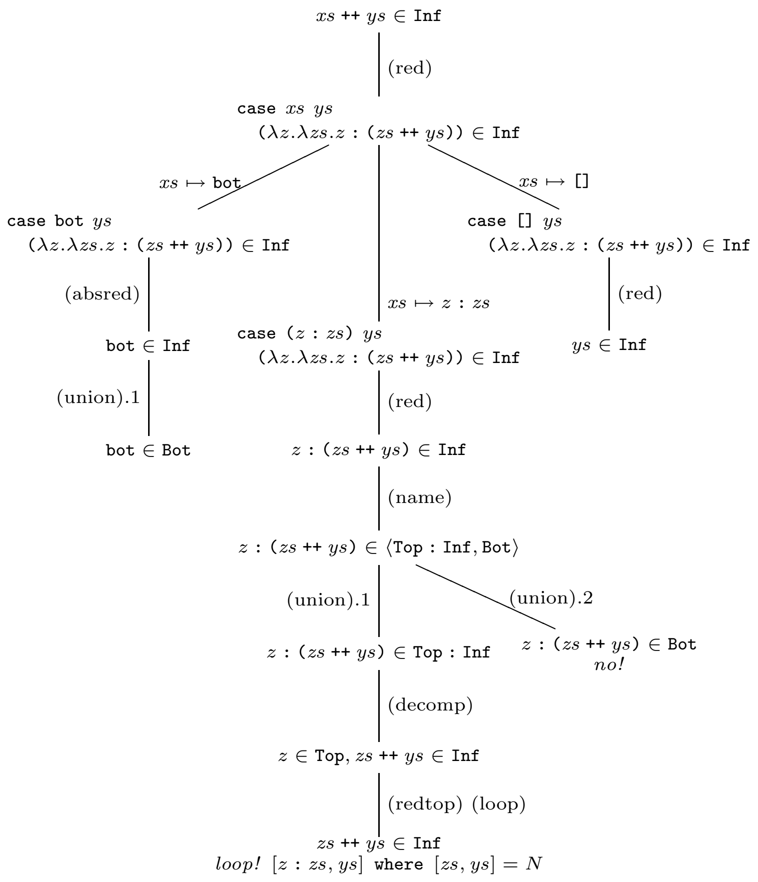

xs ++ ys ∈ Inf
(red)
case xs ys (λz.λzs.z : (zs ++ ys)) ∈ Inf
xs ↦ bot
xs ↦ z : zs
xs ↦ []
case bot ys (λz.λzs.z : (zs ++ ys)) ∈ Inf
case (z : zs) ys (λz.λzs.z : (zs ++ ys)) ∈ Inf
case [] ys (λz.λzs.z : (zs ++ ys)) ∈ Inf
(absred)
bot ∈ Inf
(union).1
bot ∈ Bot
(red)
ys ∈ Inf
(red)
z : (zs ++ ys) ∈ Inf
(name)
z : (zs ++ ys) ∈ ⟨Top : Inf, Bot⟩
(union).1
(union).2
z : (zs ++ ys) ∈ Top : Inf
z : (zs ++ ys) ∈ Bot
no!
(decomp)
z ∈ Top, zs ++ ys ∈ Inf
(redtop) (loop)
zs ++ ys ∈ Inf
loop! [z : zs, ys] where [zs, ys] = N

Figure 6.2: Closed tableau for xs `++` $ys \in$ `Inf`

⟨[`Bot`, `Top`], [`[]`, `Inf`], [`Top` : zs, ys] `where` $[zs, ys] = N$⟩ as its standard representation $\mathcal{T}_N$.

Note that CADE's rules suffice to close this tableau and hence it is externally sound and complete.

In order to use this result in the remaining analysis, we will simplify it according to example 3.116. So $N \equiv_\gamma$ ⟨[`Inf`, `Top`], [`Fin`, `Inf`]⟩.

Now we have set everything up for the analysis of `foldr (++) []` $zs \in$ `Inf` presented in figure 6.3.

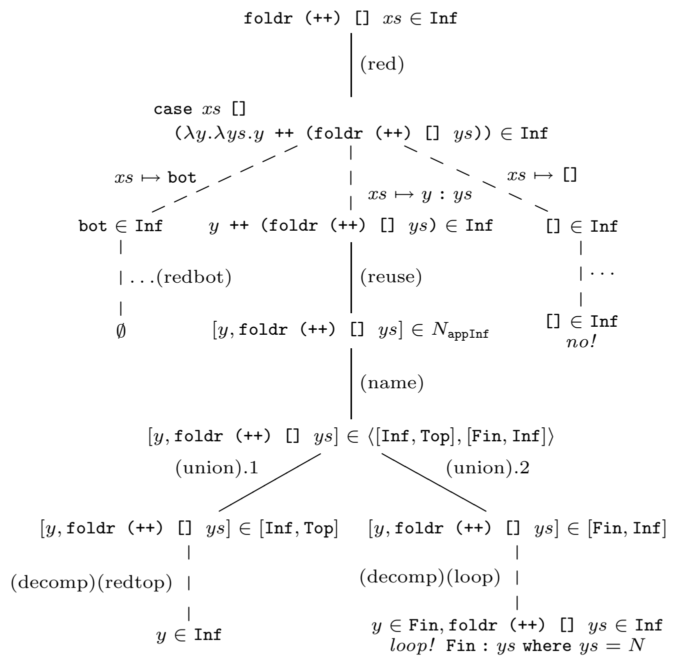

foldr (++) [] xs ∈ Inf
(red)
case xs [] (λy.λys.y ++ (foldr (++) [] ys)) ∈ Inf
xs ↦ bot
xs ↦ y : ys
xs ↦ []
bot ∈ Inf
...(redbot)
∅
y ++ (foldr (++) [] ys) ∈ Inf
(reuse)
[y, foldr (++) [] ys] ∈ N_appInf
[] ∈ Inf
...
[] ∈ Inf
no!
(name)
[y, foldr (++) [] ys] ∈ ⟨[Inf, Top], [Fin, Inf]⟩
(union).1
(union).2
[y, foldr (++) [] ys] ∈ [Inf, Top]
[y, foldr (++) [] ys] ∈ [Fin, Inf]
(decomp)(redtop)
y ∈ Inf
(decomp)(loop)
y ∈ Fin, foldr (++) [] ys ∈ Inf
loop! Fin : ys where ys = N

Figure 6.3: Closed tableau for `foldr (++) []` $xs \in$ `Inf`

From the tableau in figure 6.3, $\mathcal{T}'$, we collect the standard representation $\mathcal{T}'_{N'}$ defining $N' = \langle$`Bot`, `Inf : Top`, `Fin :` ys `where` $ys = N'\rangle$.

The (reuse) rule is necessary in the sense that without it or similar rules this tableau could not have been closed.

6.1.2 Expanding demand names late

This section addresses one of many possible heuristics that affect an implementation's efficiency. Much more work in this direction is needed to arrive at a production quality implementation of ADE and CADE.
It seems to be beneficial to the size of the tableau to delay demand name expansion.
A good example where the benefits can be seen is the analysis of `f` x y $\in$ `Fin` where `Fin` = $\langle$`[], Top:Fin`$\rangle$ and `f` is defined as follows:

```
f x y  = case x y (f' y)
f' x n y = f x y
```

The analysis proceeds as shown in figure 6.4.
Expanding the demand name, `Inf`, early and decomposing the union, gives a tableau with essentially twice as many nodes.

6.1.3 Simple higher-order

There are some higher-order constraints which our calculi will analyze without even applying the higher-order rules from section 5.3. This is possible because due to the "analyze-by-need" character of the calculi an intermediate higher-order expression may

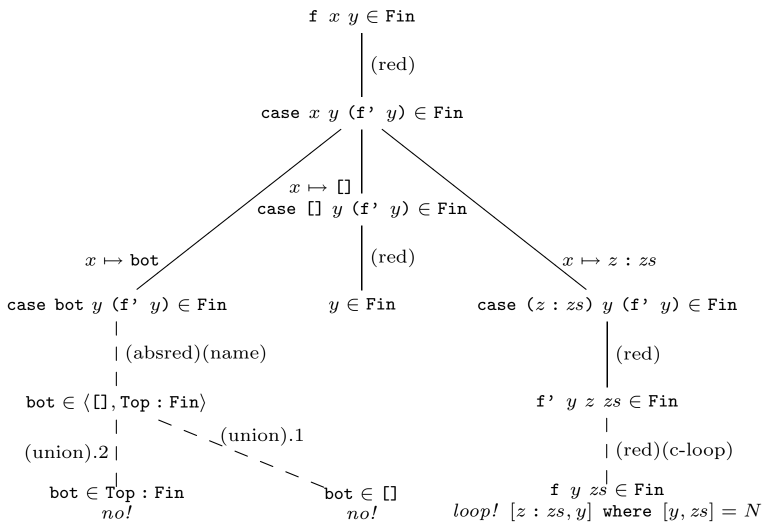


Figure 6.4: Late name expansion to keep tableau small

e.g. be constrained by `Top` and can in this case be eliminated by the (redtop) rule. An example analysis can be found in figure 6.5.

6.1.4 Negative Examples

In this section we will discuss some examples on which our implementation fails to produce the complete result. The first is one that might indeed be remedied by formulating the initial constraint differently.

Flattening Trees

Let a data type with two constructors for trees be given: one for the empty tree and one for a binary branch node containing data.

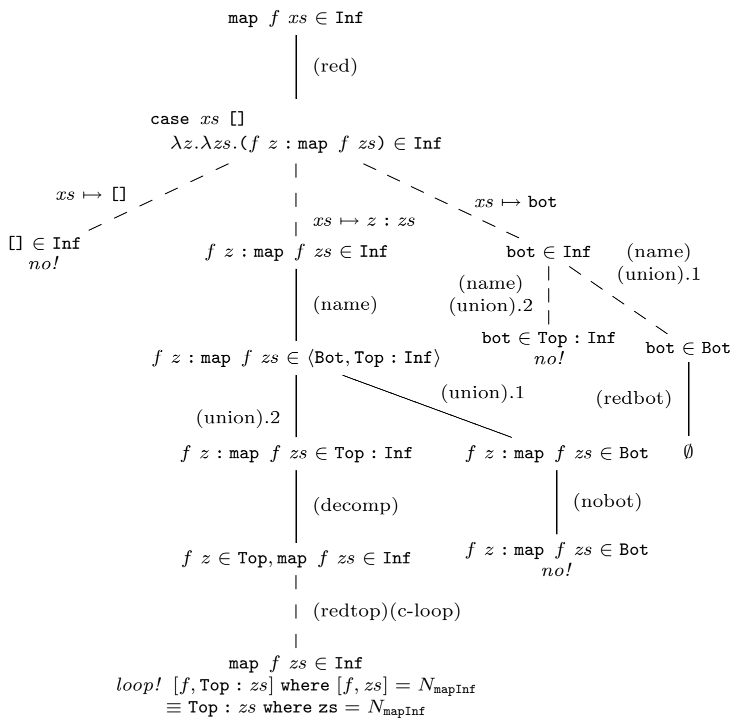

map f $xs \in$ Inf
(red)
case xs []
$\lambda z.\lambda zs.(f\ z$: map $f\ zs) \in$ Inf
$xs \mapsto$ []
[] $\in$ Inf
no!
$xs \mapsto z : zs$
$xs \mapsto$ bot
$f\ z$: map $f\ zs \in$ Inf
bot $\in$ Inf
(name)
(union).1
(name)
(union).2
(name)
bot $\in$ Top : Inf
no!
bot $\in$ Bot
$f\ z$: map $f\ zs \in \langle$Bot, Top : Inf$\rangle$
(union).1
(redbot)
(union).2
$f\ z$: map $f\ zs \in$ Top : Inf
$f\ z$: map $f\ zs \in$ Bot
$\emptyset$
(decomp)
(nobot)
$f\ z \in$ Top, map $f\ zs \in$ Inf
$f\ z$: map $f\ zs \in$ Bot
no!
(redtop)(c-loop)
map $f\ zs \in$ Inf
loop! $[f$, Top : $zs]$ where $[f, zs] = N_{\texttt{mapInf}}$
$\equiv$ Top : zs where zs $= N_{\texttt{mapInf}}$

Figure 6.5: Simple higher-order handled by base calculi

The function `flatten` maps such trees to the list of their data corresponding to an in-order traversal of the tree.

```
flatten ts = case_Tree ts [] flatc

flatc x t1 t2
  = case_Tree t1 (x : (flatten t2)) (flatc' x t2)
```

```
flatc' x t2 y t'1 t'2
  = flatten (Br y t'1 (Br x t'2 t2))
```

An open tableau from this analysis is shown in figure 6.6.

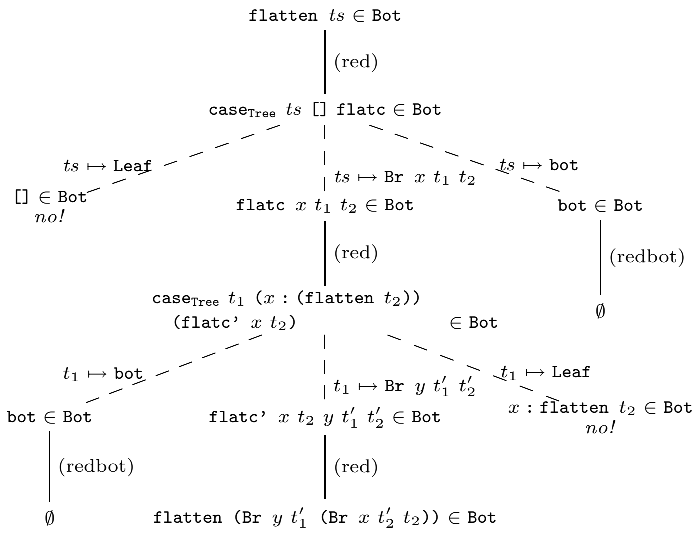

flatten $ts \in$ Bot
(red)
case$_{\texttt{Tree}}$ ts [] flatc $\in$ Bot
$ts \mapsto$ Leaf
$ts \mapsto$ Br $x\ t_1\ t_2$
$ts \mapsto$ bot
[] $\in$ Bot
no!
flatc $x\ t_1\ t_2 \in$ Bot
bot $\in$ Bot
(red)
(redbot)
case$_{\texttt{Tree}}$ t_1 (x : (flatten t_2)) (flatc' $x\ t_2$) $\in$ Bot
$\emptyset$
$t_1 \mapsto$ bot
$t_1 \mapsto$ Br $y\ t_1'\ t_2'$
$t_1 \mapsto$ Leaf
bot $\in$ Bot
flatc' $x\ t_2\ y\ t_1'\ t_2' \in$ Bot
x : flatten $t_2 \in$ Bot
no!
(redbot)
(red)
$\emptyset$
flatten (Br $y\ t_1'$ (Br $x\ t_2'\ t_2$)) $\in$ Bot

Figure 6.6: Open tableau for `flatten` $ts \in$ `Bot`

No loop rule can be applied: the condition $size(\sigma\vec{x}) > size(\vec{t})$ is violated and so is $\sigma\vec{x} \equiv \vec{t}$.
Formulated as above, `flatten` is one of the simplest examples for which a simple order of expression sizes is insufficient to close the tableau and for which the size of different arguments will need to be weighed differently, e.g. giving the first weight two and the second weight one. In our prototypical implementation we use one simple built-in size function, but this could be replaced by expression orders specifically tailored for individual analyses. This

approach has successfully been explored for termination analysis of non-strict functional programming languages in [Pan97] and for termination analysis of strict functional languages in e.g. [Gie95]. But `flatten` may also be formulated as below.

$$\begin{aligned} \texttt{flatten}\ ts &= \texttt{case}_{Tree}\ ts\ \texttt{[]}\ \texttt{flatc} \\ \texttt{flatc}\ x\ t_1\ t_2 &= (\texttt{flatten}\ t_1)\ \texttt{++}\ ([x]\ \texttt{++}\ (\texttt{flatten}\ t_2)) \end{aligned}$$

In the analysis of $\texttt{flatten}\ ts \in \texttt{Bot}$ we will need the results of the analysis for $\texttt{append}\ xs\ ys \in \texttt{Bot}$. Refer to figure 6.7 for a closed tableau from this analysis.

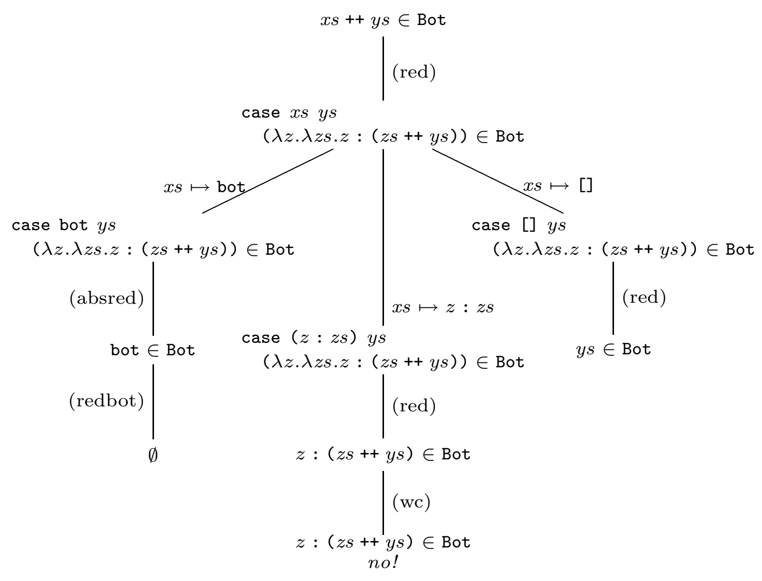

xs ++ ys ∈ Bot
(red)
case xs ys (λz.λzs.z : (zs ++ ys)) ∈ Bot
xs ↦ bot
xs ↦ []
xs ↦ z : zs
case bot ys (λz.λzs.z : (zs ++ ys)) ∈ Bot
(absred)
bot ∈ Bot
(redbot)
∅
case (z : zs) ys (λz.λzs.z : (zs ++ ys)) ∈ Bot
(red)
z : (zs ++ ys) ∈ Bot
(wc)
z : (zs ++ ys) ∈ Bot
no!
case [] ys (λz.λzs.z : (zs ++ ys)) ∈ Bot
(red)
ys ∈ Bot

Figure 6.7: Closed tableau for $\texttt{append}\ xs\ ys \in \texttt{Bot}$

The standard representation of this tableau is $N_{\texttt{appBot}} =$

$\langle[\texttt{Bot}, \texttt{Top}], [\texttt{[]}, \texttt{Bot}]\rangle$.

Now we are prepared for the analysis of $\texttt{flatten}\ ts \in \texttt{Bot}$ as presented in figure 6.8.

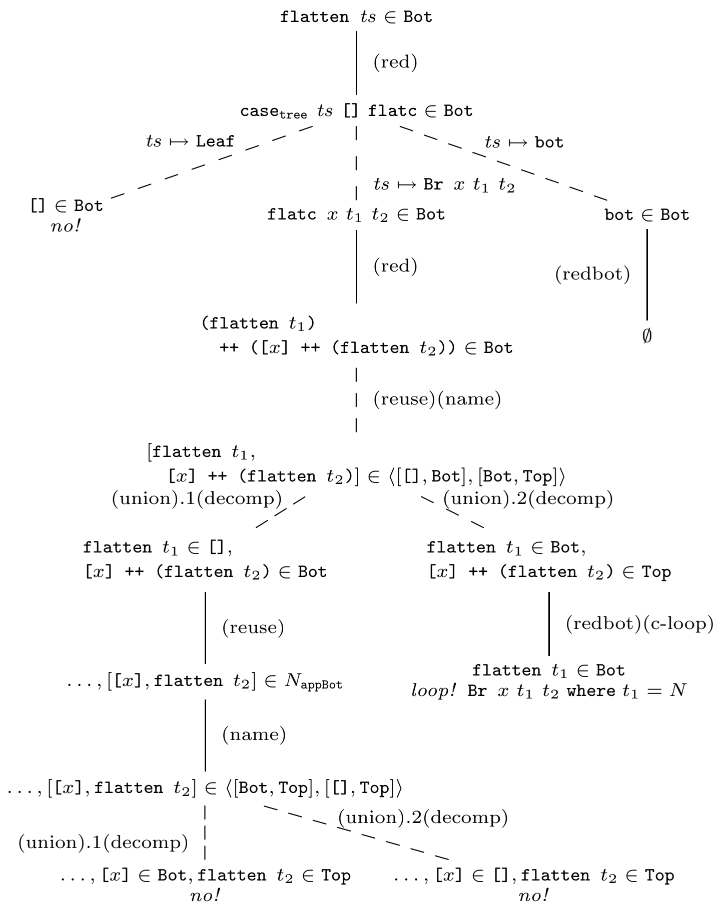

flatten ts ∈ Bot
(red)
case_tree ts [] flatc ∈ Bot
ts ↦ Leaf
ts ↦ Br x t1 t2
ts ↦ bot
[] ∈ Bot
no!
flatc x t1 t2 ∈ Bot
(red)
(flatten t1) ++ ([x] ++ (flatten t2)) ∈ Bot
(reuse)(name)
[flatten t1, [x] ++ (flatten t2)] ∈ ⟨[[], Bot], [Bot, Top]⟩
(union).1(decomp)
(union).2(decomp)
flatten t1 ∈ [], [x] ++ (flatten t2) ∈ Bot
(reuse)
..., [[x], flatten t2] ∈ N_appBot
(name)
..., [[x], flatten t2] ∈ ⟨[Bot, Top], [[], Top]⟩
(union).1(decomp)
(union).2(decomp)
..., [x] ∈ Bot, flatten t2 ∈ Top
no!
..., [x] ∈ [], flatten t2 ∈ Top
no!
flatten t1 ∈ Bot, [x] ++ (flatten t2) ∈ Top
(redbot)(c-loop)
flatten t1 ∈ Bot
loop! Br x t1 t2 where t1 = N
bot ∈ Bot
(redbot)
∅

Figure 6.8: Closed tableau for $\texttt{flatten}\ ts \in \texttt{Bot}$

The result of this analysis is `flatten` $ts \in$ `Bot` $\rightarrow_{\text{ADE}} N$ where $N = \langle$`Bot`, `Br` $x\ t_1\ t_2$ `where` $t_1 = N\rangle$. This result could be shortened to `FL` $=$ $\langle$`Bot`, `Br Top FL Top`$\rangle$ provided the demand name `FL` is previously unused.

Missing rules

We present a tableau which one might hope to obtain for an analysis involving `copy`. The calculi are missing a rule to produce this tableau and we give the intuition for this missing rule.
The function `copy` can be defined as below

```
copy n x = take n (repeat x)
take n x = case n [] (take' x)
take' x m = case x [] (take'' m)
take'' m x y = x:(take m y)
```

We want to analyze `copy` $n\ x \in$ `Inf` so we will analyze `take` $n\ xs \in$ `Inf` first and present closed tableau of the latter analysis in figure 6.9.
The result is

$$N_{\texttt{takeInf}} = \langle [\texttt{Bot}, \texttt{Top}], [\texttt{Succ Top}, \texttt{Bot}], \\ [\texttt{Succ}\ r, ys]\ \texttt{where}\ [r, ys] = N_{\texttt{takeInf}}\rangle.$$

We use this result in the hypothetical tableau for `copy` $n\ x \in$ `Inf` shown in figure 6.10.
There it can be seen that we would have needed to start with root $[n, \texttt{repeat}\ x] \in N_{\texttt{takeInf}}$ in order to close loops with the root. Obviously, this would be easy to fix. Much harder, though surely not impossible to fix, is the missing rule above the (redwhere) rule. Intuitively, this rule would need to "commit" a free variable

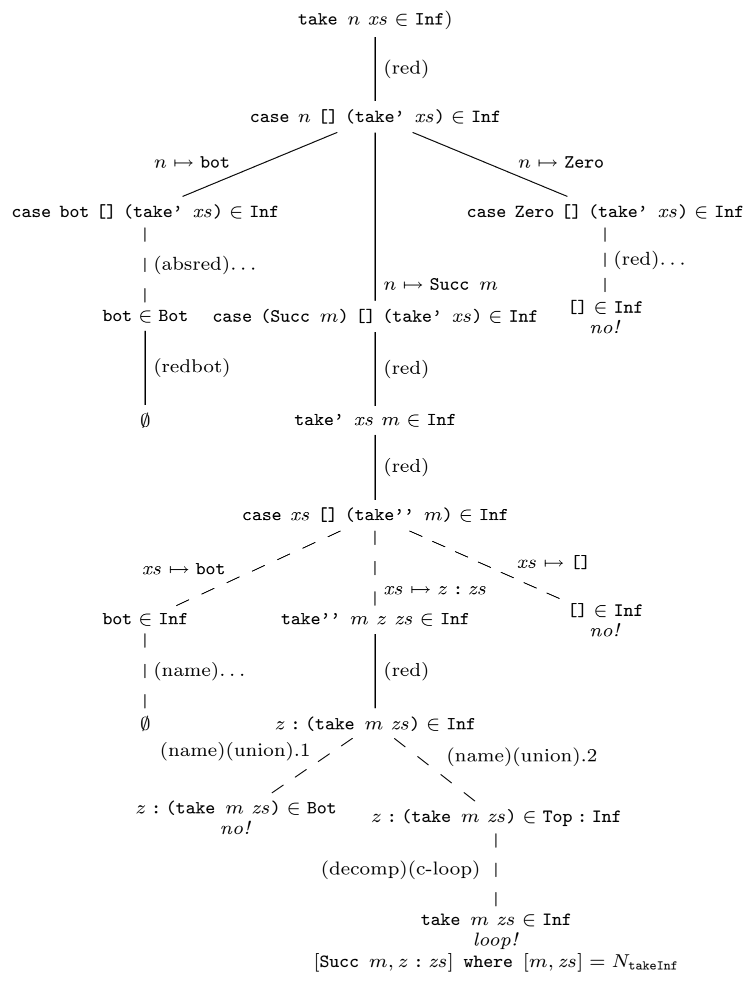

take n xs ∈ Inf)
(red)
case n [] (take' xs) ∈ Inf
$n \mapsto$ bot
$n \mapsto$ Zero
case bot [] (take' xs) ∈ Inf
case Zero [] (take' xs) ∈ Inf
(absred)...
$n \mapsto$ Succ m
(red)...
[] ∈ Inf
no!
bot ∈ Bot
case (Succ m) [] (take' xs) ∈ Inf
(redbot)
(red)
∅
take' xs m ∈ Inf
(red)
case xs [] (take'' m) ∈ Inf
$xs \mapsto$ bot
$xs \mapsto$ []
$xs \mapsto z : zs$
bot ∈ Inf
take'' m z zs ∈ Inf
[] ∈ Inf
no!
(name)...
(red)
∅
z : (take m zs) ∈ Inf
(name)(union).1
(name)(union).2
z : (take m zs) ∈ Bot
no!
z : (take m zs) ∈ Top : Inf
(decomp)(c-loop)
take m zs ∈ Inf
loop!
[Succ $m, z : zs$] where $[m, zs] = N_{\texttt{takeInf}}$

Figure 6.9: Closed tableau for `take` $n\ xs \in$ `Inf`

in the expression of a constraint to be formed with a particular constructor, much as the (casep) does for the variable being cased. Here, the first component of any expression in the concretization of $[\texttt{Succ}\ r, ys]$ `where` $[r, ys] = N_{\texttt{takeInf}}$ will be formed with the `Succ`-constructor so the variable n can be committed to have `Succ` as its top level constructor.

6.2 Primitives

This section considers a larger example demonstrating analyses that can be performed by our calculi as well as some that go beyond their limits. This example will be the integers, $\mathbb{Z}$, and their arithmetic operations. Commonly, integers are provided as a *primitive* data type in programming languages. The arithmetic operations are usually hyper-strict and the primitive data type covers only an interval of the integers.

This example uses the syntax of our prototypical implementation, which is different from the syntax of Λ. In particular, it uses super-combinators. It is very similar to the syntax of [Sch94, PJL91]. As was noted in section 2.5.8, super-combinators can straightforwardly be transformed into Λ-expressions.

The integers are encoded as signed Peano numbers. Figure 6.11 shows the definitions for Peano numbers.

Based on these definitions we define the functions for the signed Peano numbers in figure 6.12.

The calculi are able to analyse e.g. `addI` $x\ y \in$ `2` where `2` is a shorthand notation for `Pos (Succ (Succ Zero))`. In this analysis some analysis results from other analyses need to be reused. These other analyses are: `subPI` $x\ y \in$ `2` and `addP` $x\ y \in$ `Succ (Succ Zero)`. The analysis of the latter furthermore reuses

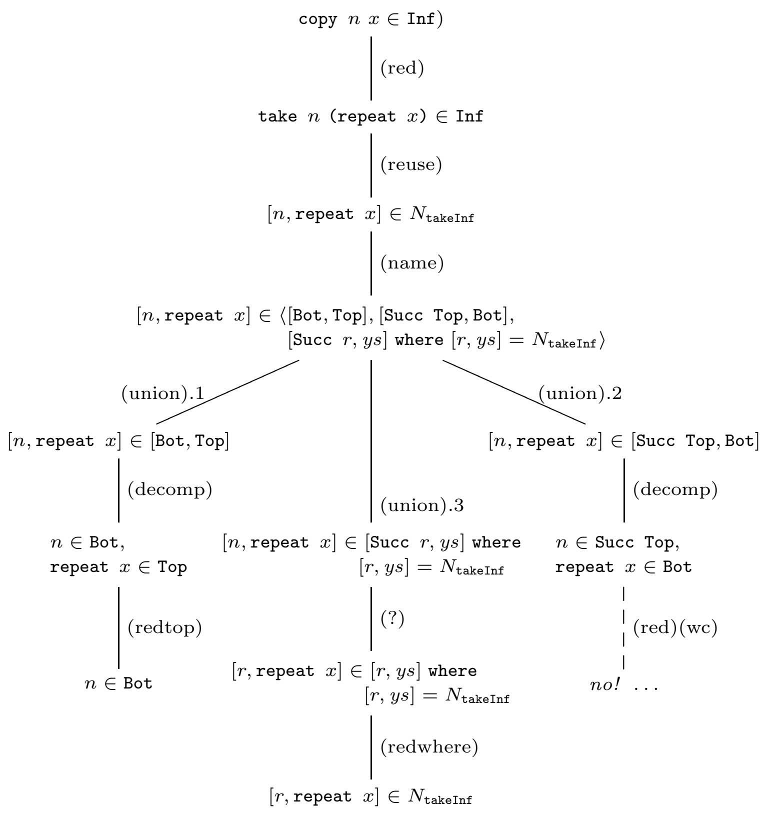

copy n x ∈ Inf)
(red)
take n (repeat x) ∈ Inf
(reuse)
[n, repeat x] ∈ N_takeInf
(name)
[n, repeat x] ∈ ⟨[Bot, Top], [Succ Top, Bot], [Succ r, ys] where [r, ys] = N_takeInf⟩
(union).1
(union).2
[n, repeat x] ∈ [Bot, Top]
[n, repeat x] ∈ [Succ Top, Bot]
(decomp)
(union).3
(decomp)
n ∈ Bot, repeat x ∈ Top
[n, repeat x] ∈ [Succ r, ys] where [r, ys] = N_takeInf
n ∈ Succ Top, repeat x ∈ Bot
(redtop)
(?)
(red)(wc)
n ∈ Bot
[r, repeat x] ∈ [r, ys] where [r, ys] = N_takeInf
no! ...
(redwhere)
[r, repeat x] ∈ N_takeInf

Figure 6.10: Missing rule in tableau for `copy` $n\ x \in$ `Inf`

the result of `addP` $x\ y \in$ `Succ Zero`. For this last analysis we do not provide a closed tableau, but state its result $N_{\texttt{addPSuccZero}} = \langle[\texttt{Zero}, \texttt{Succ Zero}], [\texttt{Succ Zero}, \texttt{Zero}]\rangle$. A closed tableau for the analysis of `addP` $x\ y \in$ `Succ (Succ Zero)` is shown in figure 6.13, one for `subPI` $x\ y \in$ `2` in figure 6.14, and one for `addI` $x\ y \in$ `2` in

```
Zero = Cons{1,0};
Succ = Cons{2,1};

strictPeano f x
 = case x of
    <1> -> f Zero;
    <2> y -> f (Succ y);

hsPeano x
 = case x of
    <1> -> Zero;
    <2> y -> strictPeano Succ (hsPeano y);

addP x y
 = case x of
    <1> -> y;
    <2> z -> Succ (addP z y);
```

Figure 6.11: Peano numbers and their operations

figure 6.15.

It seems natural to analyze `add` $x\ y \in 2$ and figure 6.16 shows an open tableau from this analysis.

The bottom leaf in this figure is $\{$`hsPeano` $r \in$ `Succ` $\dot{z}$, `hsPeano` $s \in$ `Succ` $\dot{a}, [z, a] \in [\dots]$ `where` $\dots\}$. Even with all extensions from chapter 5 a tableau for e.g. `hsPeano` $r \in$ `Succ` $\dot{z}$ cannot be closed as is shown in figure 6.17.

At first it may seem that only the presence of the demand variable prevents closing the paths and that if this variable would be bound

```
Neg = Cons{3,1};
Pos = Cons{4,1};

subPI x y
 = case x of
    <1> -> (case y of
             <1> -> Pos Zero;
             <2> s -> Neg y);
    <2> r -> case y of
            <1> -> Pos x;
            <2> s -> subPI r s;

addI x y
 = case x of
    <3> r -> (case y of
               <3> s -> Neg (addP r s);
               <4> s -> subPI s r);
    <4> r -> case y of
              <3> s -> subPI r s;
              <4> s -> Pos (addP r s);

subI x y
 = case y of
    <3> s -> addI x (Pos s);
    <4> s -> case x of
              <3> r -> Neg (addP r s);
              <4> r -> subPI r s;

hsIntP x
 = case x of
    <3> r -> strictPeano Neg (hsPeano r);
    <4> r -> strictPeano Pos (hsPeano r);

add x y
 = addI (hsIntP x) (hsIntP y);

sub x y
 = subI (hsIntP x) (hsIntP y);
```

Figure 6.12: Operations on signed Peano numbers

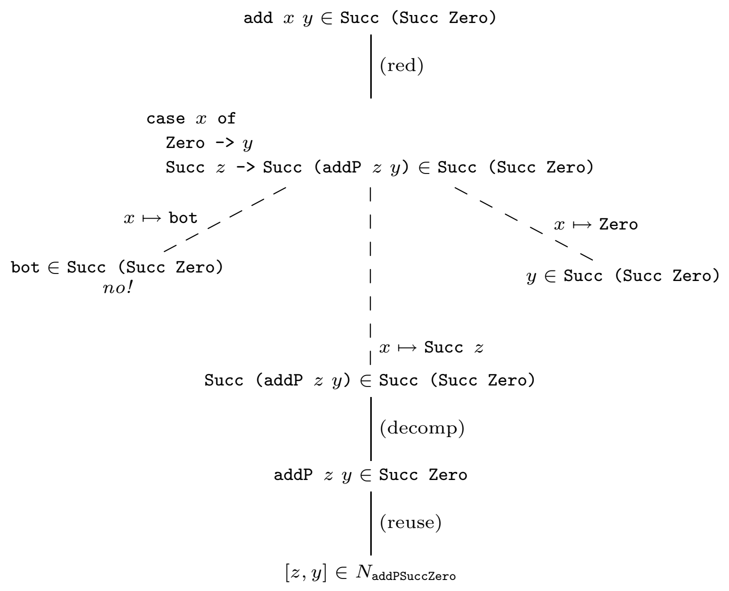

add x $y \in$ Succ (Succ Zero)
(red)
case x of
Zero -> y
Succ z -> Succ (addP z y) $\in$ Succ (Succ Zero)
$x \mapsto$ bot
bot $\in$ Succ (Succ Zero)
no!
$x \mapsto$ Zero
$y \in$ Succ (Succ Zero)
$x \mapsto$ Succ z
Succ (addP z y) $\in$ Succ (Succ Zero)
(decomp)
addP z $y \in$ Succ Zero
(reuse)
$[z, y] \in N_{\texttt{addPSuccZero}}$

Figure 6.13: Closed tableau for `addP` x $y \in$ `Succ (Succ Zero)`.

to an appropriate constructor expression a result for the analysis could be obtained. But the two nodes $\{y \in \texttt{Zero}, \texttt{Zero} \in \dot{c}\}$ and $\{\texttt{hsPeano}\ y \in \texttt{Succ}\ \dot{z}, \texttt{Succ}\ z \in \dot{c}\}$ would require different bindings for the demand variable $\dot{c}$ and it is not at all obvious how to obtain solutions for the root constraint in this situation. How to proceed in this situation will need to be clarified by future work.

Alternatively, one could try to analyse the root constraints $\{\texttt{hsPeano}\ r \in \texttt{Succ}\ \dot{z}, \texttt{hsPeano}\ s \in \texttt{Succ}\ \dot{a}, [z, a] \in N_{\texttt{subPI2}}\}$ and in this analysis a (commit) rule as intuitively motivated in section 6.1.4 would be needed to arrive at a node $\{\texttt{hsPeano}\ rr \in$

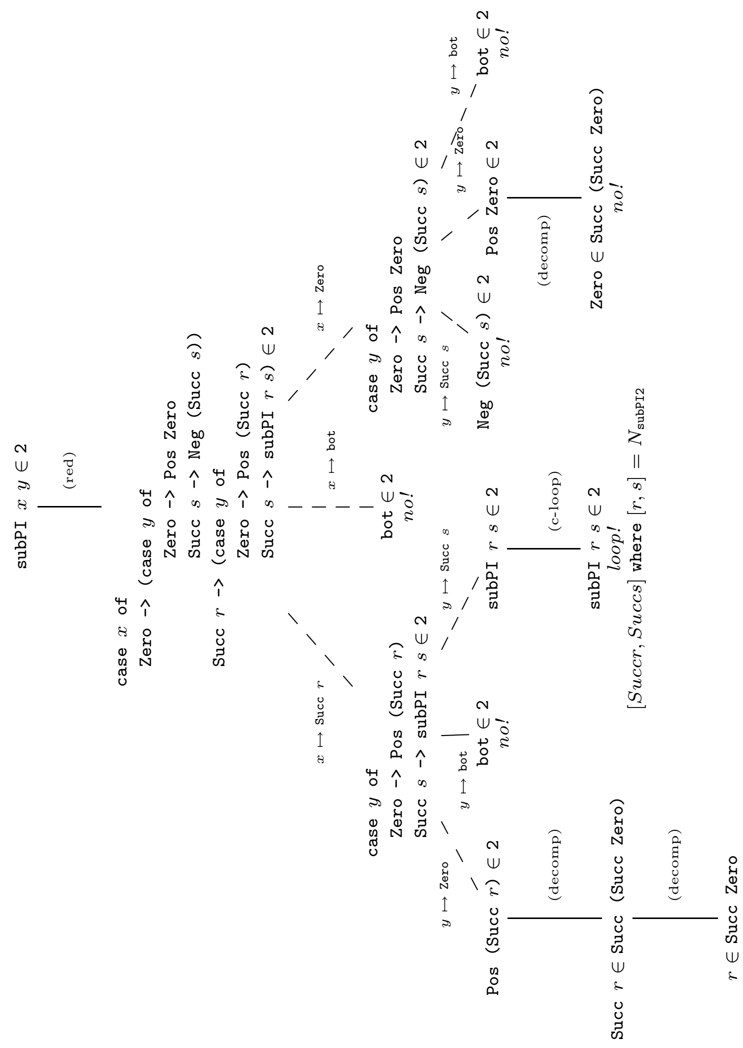


Figure 6.14: Closed tableau for `subPI` x $y \in 2$.

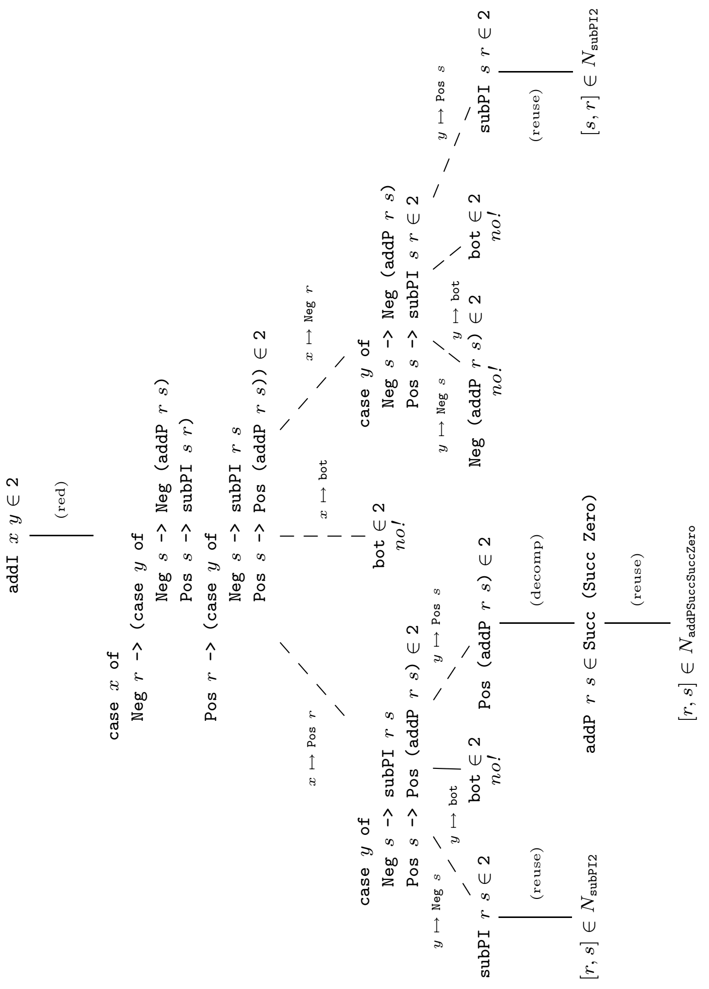


Figure 6.15: Closed tableau for `addI` x $y \in 2$.

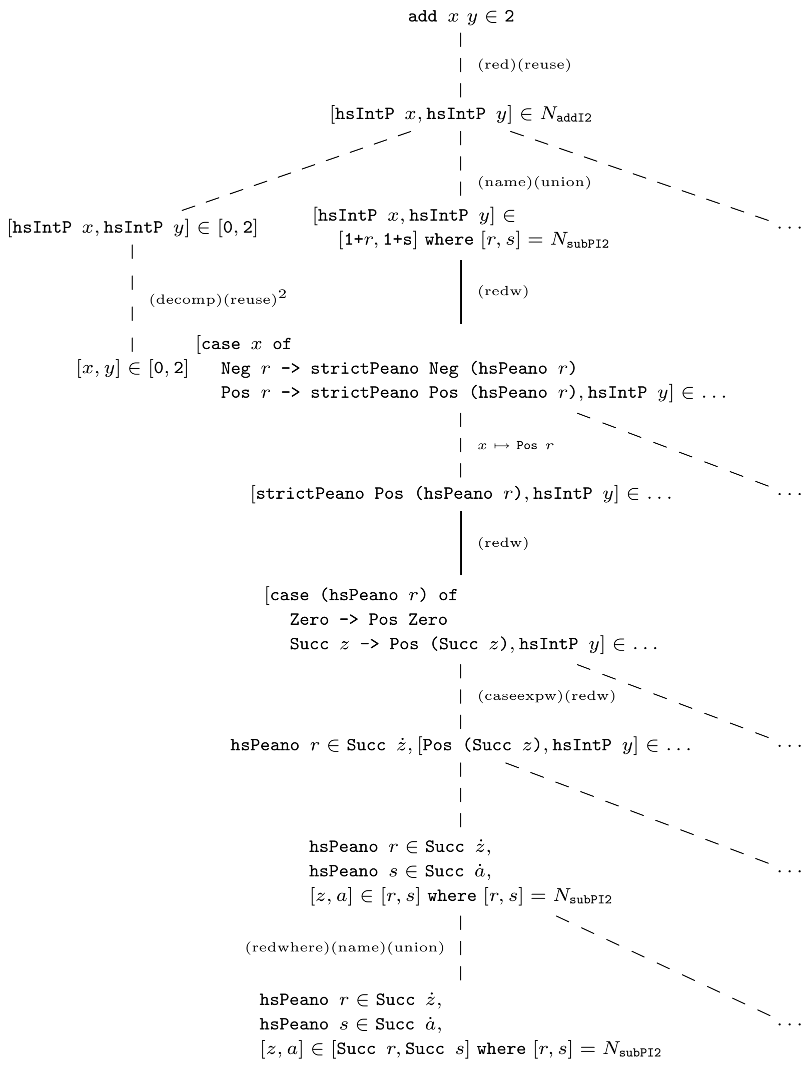

add x $y \in 2$
(red)(reuse)
[hsIntP x, hsIntP y] $\in N_{\texttt{addI2}}$
(name)(union)
[hsIntP x, hsIntP y] $\in$ [0, 2]
(decomp)(reuse)^2
$[x, y] \in [0, 2]$
[hsIntP x, hsIntP y] $\in$
[1+r, 1+s] where $[r, s] = N_{\texttt{subPI2}}$
(redw)
[case x of
Neg r -> strictPeano Neg (hsPeano r)
Pos r -> strictPeano Pos (hsPeano r), hsIntP y] $\in \ldots$
$x \mapsto$ Pos r
[strictPeano Pos (hsPeano r), hsIntP y] $\in \ldots$
(redw)
[case (hsPeano r) of
Zero -> Pos Zero
Succ z -> Pos (Succ z), hsIntP y] $\in \ldots$
(caseexpw)(redw)
hsPeano $r \in$ Succ $\dot{z}$, [Pos (Succ z), hsIntP y] $\in \ldots$
hsPeano $r \in$ Succ $\dot{z}$,
hsPeano $s \in$ Succ $\dot{a}$,
$[z, a] \in [r, s]$ where $[r, s] = N_{\texttt{subPI2}}$
(redwhere)(name)(union)
hsPeano $r \in$ Succ $\dot{z}$,
hsPeano $s \in$ Succ $\dot{a}$,
$[z, a] \in$ [Succ r, Succ s] where $[r, s] = N_{\texttt{subPI2}}$
$\ldots$

Figure 6.16: Open tableau for `add` x $y \in 2$.

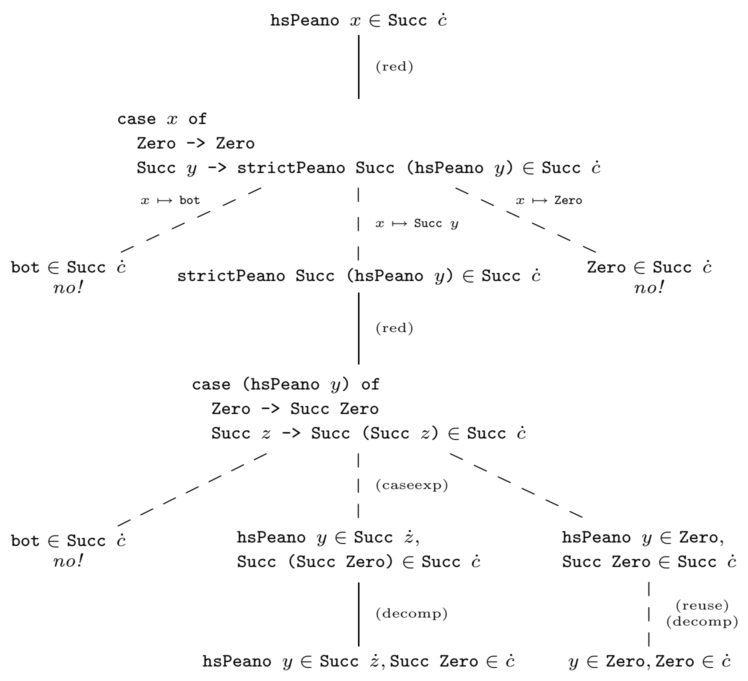

hsPeano x ∈ Succ $\dot{c}$
(red)
case x of
Zero -> Zero
Succ y -> strictPeano Succ (hsPeano y) ∈ Succ $\dot{c}$
$x \mapsto$ bot
$x \mapsto$ Succ y
$x \mapsto$ Zero
bot ∈ Succ $\dot{c}$
no!
strictPeano Succ (hsPeano y) ∈ Succ $\dot{c}$
Zero ∈ Succ $\dot{c}$
no!
(red)
case (hsPeano y) of
Zero -> Succ Zero
Succ z -> Succ (Succ z) ∈ Succ $\dot{c}$
(caseexp)
bot ∈ Succ $\dot{c}$
no!
hsPeano y ∈ Succ $\dot{z}$,
Succ (Succ Zero) ∈ Succ $\dot{c}$
hsPeano y ∈ Zero,
Succ Zero ∈ Succ $\dot{c}$
(decomp)
(reuse)
(decomp)
hsPeano y ∈ Succ $\dot{z}$, Succ Zero ∈ $\dot{c}$
y ∈ Zero, Zero ∈ $\dot{c}$

Figure 6.17: Open tableau for `hsPeano` $x \in$ `Succ` $\dot{c}$.

`Succ` $\dot{z}z$, `hsPeano` $ss \in$ `Succ` $\dot{a}a, [zz, aa] \in N_{\texttt{subPI2}}\}$. Here also, future work will be necessary, since currently global rules are applicable only if the root is a single constraint.

6.3 Applications

Demand analysis is a versatile tool. A variety of analyses can be performed by encoding into an appropriate constraint for demand analysis.

6.3.1 Evaluation degree

Some demands can be interpreted as a degree of evaluation and then ADE and CADE determine the evaluation transformers of a given Λ-expression. There are two essentially differing possibilities for this interpretation.

We can interpret a demand as the degree of evaluation necessary to determine that a closed Λ-expression is a member of the demand's concretization. Then e.g. $\langle$`True`, `False`$\rangle$ stands for an evaluation to WHNF if $\{$`True`, `False`$\} =$ `Bool` $\in \mathcal{A}$. We might be tempted to force evaluation to WHNF if we find the demand for an argument of an application to be $\langle$`True`, `False`$\rangle$. However, ADE is externally sound but not always externally complete and the complete demand for the argument might be `Top` and no evaluation would be necessary, i.e. the semantics of the program might be changed based on this result. Thus this interpretation is safe only for CADE, but not for ADE.

The dual interpretation, i.e. that a demand stands for the degree of evaluation necessary to determine that a closed Λ-expression is *not* a member of the demand's concretization, is safe both for ADE and for CADE. Then `Bot` stands for evaluation to WHNF, `Top` stands for no evaluation at all since all closed Λ-expressions are members of its concretization, and `Inf` $= \langle$`Bot`, `Top : Inf`$\rangle$ stands for evaluation to WHNF as well as evaluation to the same degree as the original expression for the tail if that WHNF uses the `:`-constructor, i.e. spine-strict evaluation. With this interpretation demands such as `True` do not correspond to evaluation degrees since none of the expressions $\equiv_c$-equivalent to `bot` is in the concretization of `True` and no amount of evaluation can be sure to recognize all non-terminating computations. Furthermore, this allows interpretation of demands such as `Bot : Top` as a "delayed evaluation degree": while no evaluation degree can be associated with the entire demand, as soon as an expression has been evaluated to s_1 : s_2 its head needs to be evaluated to WHNF. Not all approaches to strictness analysis are able to represent this kind of strictness, cf. [BHA85, Pat96].

Example 6.2. In section 6.1.1 we analyzed `foldr (++) []` $zs \in$ `Inf` to obtain the standard representation $\mathcal{T}'_{N'}$ defining $N' = \langle \texttt{Bot}, \texttt{Inf} : \texttt{Top}, \texttt{Fin} : ys\ \texttt{where}\ ys = N' \rangle$.
How can we interpret this result with respect to the degrees of evaluation of the expressions?
We have already interpreted `Inf` as spine-strict evaluation. N' can be interpreted as evaluation to WHNF, and, if a `:`-constructor results, spine-strict evaluation of its head without evaluation of its tail. As soon as the spine-strict evaluation of this head terminates due to a `[]`-constructor in the head's spine the tail of the entire expression needs to be evaluated just as this entire expression was evaluated. Intuitively, it is clear that all the lists contained in the outer list can be evaluated spine-strictly (possibly in parallel) since every single one of them needs to be finite for the spine-strict evaluation of the entire application to terminate.

Example 6.3. As an example involving jocs consider the analysis of `append` $xs\ \ ys \in$ `Inf` which demands $N = \langle [\texttt{Bot}, \texttt{Top}], [\texttt{[]}, \texttt{Inf}], [\texttt{Top} : zs, ys]\ \texttt{where}\ [zs, ys] = N \rangle$ completely. We have already shown this to be $\equiv_\gamma$-equivalent to $\langle [\texttt{Inf}, \texttt{Top}], [\texttt{Fin}, \texttt{Inf}] \rangle$ in example 3.116.
How can we conclude that both arguments can be evaluated spine-strictly from this analysis result? Since `Top` is present as the right component of a joc having `Inf` as its left component, we can conclude that no evaluation of ys needs to be performed as long

as xs can be evaluated spine strictly to `:`-constructors and this evaluation continues. As soon as `[]` is encountered as the tail, xs is in `Fin` and ys needs to be evaluated spine-strictly.

6.3.2 Runtime errors

There are different ways to conceive of runtime errors as in the definition of

$$\texttt{head} \ = \lambda xs.\texttt{case}_{\text{List}}\ xs\ \texttt{(error "head of empty list")}\ \texttt{K}.$$

One can either identify program errors with `bot` or one can identify a program error as data. Each possibility reflects a different perspective from which to view the program.
In the former case one would define

$$\texttt{error}\ \ x = \texttt{bot}$$

or

$$\texttt{head}\ \ xs = \texttt{case}_{\text{List}}\ xs\ \texttt{bot}\ \texttt{K}.$$

This reflects the intuition, that an error message and program abort does not provide any, at least not any desired, information.
For the latter case one would transform the program so that an application of `error` evaluated to some unique data value. This can be achieved by adding a constructor $\texttt{c}_{A,\texttt{error}}$ for every type constructor A, and reflects the intuition, that an error message and program abort do in fact provide more information than a program which loops and of which we do not know if it will eventually terminate.
From both perspectives we can derive sensible information even from a simple constraint such as `f` $x \in$ `Bot`, which we will now show how to interpret.

We may distinguish some cases for the result of the analysis. It may be that for the result, C, we have either $C \equiv_\gamma \emptyset, C \equiv_\gamma$ `Bot`, $C \not\equiv_\gamma$ `Bot` but `Bot` $\leq_\gamma C$, or `Bot` $\not\leq_\gamma C$ and $C \not\equiv_\gamma \emptyset$. If we identify runtime errors with `bot` and obtain $\emptyset$ from a complete analysis, we know that neither runtime error nor non-termination will be encountered no matter what the input is. I.e. the function is total and its argument is absent. Obtaining $C \equiv$ `Bot` from the analysis, we can conclude that a runtime error or non-termination in the evaluation of the argument will result in a runtime error or non-termination of the application. If this result is complete, no runtime error or non-termination will occur unless it occurs in the evaluation of the argument. This then means the programmed function is total (and strict).
Only the two results $C \equiv_\gamma \emptyset$ and $C \equiv_\gamma$ `Bot` from a complete analysis of the above constraint are acceptable. In other words: If, for example, our analysis obtains a complete result for which `0` $\leq_\gamma C$ then there is a defect in the function being analyzed, because there are data values for which this function introduces non-termination or a runtime error.
While surely only very simple defects can be detected by such analyses, these simple defects do have farreaching effects. We cite the incident of the USS Yorktown here only to demonstrate just how farreaching these effects can be. We do not, however, want to imply that the incident would have been avoided by the use of one of our analyses, but merely state that an analysis could have helped detect this particular defect. The example is arbitrarily chosen, there would have been a number of other well documented similar incidents to choose from e.g. [Kor, LLF+96, Bab97].

Example 6.4. The USS Yorktown represents one of the first so-called smart ships, on which information technology is intensively

employed in order to reduce crew. Obviously, in such an environment warranting optimal functionality of the entire system requires all components to maintain functionality.
It is much less obvious that a failure in one component may cause the entire network to crash.
In the case of the Yorktown there are different accounts of what happened. All agree that a 0 was entered manually (as opposed to being read by a sensor) in order to correct a valve's state represented as open in the system when in reality it was closed. The accounts are quite vague about what happened as a consequence, Vice Adm. Henry Giffin stated that "the Yorktown lost control of its propulsion system, because its computers were unable to divide by the number zero [...] That caused the database to overflow and crash all LAN consoles and miniature remote terminal units [workstations, the author]" according to [Sla98].
In principle we have the same problem here as for `head`: for some particular value of the argument the program exhibits a runtime error and aborts. If we have identified runtime errors and `bot` our calculi will find the result `head` $x \in$ `Bot` $\to_{\text{ADE}} \langle$`Bot`, `[]`$\rangle$. As will be clear by now, this says that `head` will either not terminate or terminate with an error on an argument evaluating to `[]`. From this analysis result to the insight that in the consequence of an input of 0 or `[]` respectively, the entire system will cease to work there are many steps. In many of them analysis results like the above will help.

If we obtain a result C which is not `bot`-closed then the function being analyzed may introduce a runtime error or non-termination even if such behavior is not present in the argument.
If on the other hand we decide not to identify a runtime error with `bot`, we have a slightly different picture. For the case in

which C is not bot-closed, but is a complete analysis result, we know that there are cases in which for data values as arguments the application does not have a WHNF. Such cases are quite likely programming errors. If we have added constructors to stand for runtime errors we can also analyze, say `head` $xs \in \mathtt{c}_{\mathtt{List},\mathtt{error}}$ and find the result `[]`. This approach is more precise since non-termination and runtime errors can be distinguished.

6.3.3 Hoare/Dijkstra/Baber

We will now relate our approach to the approaches to Hoare [Hoa69] and Dijkstra [Dij76]. We will follow [Bab87] in our notation. Both approaches use so-called *Hoare-triples* $\{P\}\ S\ \{Q\}$, which can be interpreted as the statement that the set of inputs $\{P\}$ and the set of outputs $\{Q\}$ are related by statement S. A forward view would be that inputs satisfying $\{P\}$ to program S result in outputs satisfying $\{Q\}$. This view may of course be reversed and be interpreted as saying that in order to obtain output satisfying $\{Q\}$ from program S the input will have to satisfy $\{P\}$.
Hoare's original perspective was that a program transforms an input condition (a pre-condition) into an output condition (a post-condition), while Dijkstra's perspective was that in order to satisfy a post-condition there is a pre-condition to a program which will have to be satisfied.
Pre- and post-condition may also be seen as the sets of values which satisfy them.
It is quite obvious that we cannot represent arbitrary pre- or post-conditions as demands, but there are quite some which we can represent, in fact any such condition that a Turing-machine could check (cf. section 3.4). Obviously, some pre- or post-conditions will be more conveniently represented as demands than others,

but any characterization of this convenience will be left for future work.

Example 6.5. A demand representing all the finite lists with even elements can easily be expressed, e.g.

$$\begin{aligned}\mathtt{EvenPeanos} &= \langle \mathtt{EvenPeano} : \mathtt{EvenPeanos}, \mathtt{[]} \rangle\\ \mathtt{EvenPeano} &= \langle \mathtt{Zero}, \mathtt{Succ}\ (\mathtt{Succ}\ \mathtt{EvenPeano})\rangle.\end{aligned}$$

6.3.4 n-Packs

Often data structures are not consumed in single units, but by considering whole blocks of values in a way where it is not known beforehand whether more data needs to be consumed, but if more is consumed then an entire block will be consumed in a particular way. This way of evaluating is a generalization of head-strictness in which every block consists of exactly one list element. Obviously, we may consider some parts of data structures other than lists as blocks as well, e.g. we may consider subtrees of binary trees of a fixed height as blocks.
n-packs can be obtained, as Norbert Klose describes in his master's thesis [Klo97], by replacing the data type to be packed with a new type which in addition to the constructors present in the original type provides a constructor for a pack of n elements of the original type.

Example 6.6. For lists we would start out with the constructors `:` and `[]` and replace successive `:` by $:^n$ of arity n.

Obviously this method is partially subsumed by deforestation, which attempts to obtain *treelessness*. However, the n-pack transformation is possible in cases in which the conditions required by deforestation are not met. It would surely be interesting to compare the methods further, in particular empirically analyzing how frequently the case occurs in large programs that deforestation is not applicable but the n-pack transformation using analysis results is applicable. This will not be subject of the present work.

The program text of a function is possibly transformed into several functions, each of which trying to produce a densely packed return value provided some conditions are met. These conditions stem from case constructs, which are "in the way" for packing to proceed. An example for this would be the transformation of `append` into a function `!append` both consuming and producing lists, which in addition to the constructors `[]` and `:` use a 3-ary constructor `Pack` whose first and second argument are list elements and whose third argument is the tail following these elements. The difference between the two functions is that `!append` is obtained by unfolding the recursive application in `append`'s body, then lifting the case up from below the `:` constructor and using the `Pack` constructor where appropriate. `!append` is shown in figure 6.18.

`!append` is more strict than the original `append` straightforwardly extended to `nLists`. This can e.g. be verified with our calculi: $\texttt{append}\ xs\ ys \in \texttt{Bot} \rightarrow_{\text{CADE}} \langle[\texttt{Bot},\texttt{Top}],[\texttt{[]},\texttt{Bot}]\rangle$ and $\texttt{!append}\ xs\ ys \in \texttt{Bot} \rightarrow_{\text{CADE}} \langle[\texttt{Bot},\texttt{Top}],[\texttt{[]},\texttt{Bot}],[\texttt{Top}:\texttt{Bot},\texttt{Top}]\rangle$. This is the only difference in the behavior of `append` and `!append`, in particular if application of `!append` to some argument has a WHNF application of `append` yields the same WHNF. In order to maintain the semantics of the original program, we may only replace this new function for the original if the demand on the entire application is sufficient for the demands on the arguments to be the same before and after the replacement. Actually,

```
!append xs ys =
    case_nList xs
        ys
        (λz.λzs.case_nList zs
                    (z : ys)
                    (λv.λvs.Pack z v (!append vs ys))
                    (λv.λv'.λvs.
                      Pack z v (!append (v' : vs) ys)))
        (λz.λz'.λzs.Pack z z' (!append zs ys))
```

Figure 6.18: Definition of `!append`

for a safe replacement it will suffice if the demand on the entire application is sufficient for the demands after replacement to be at most as strong as before. The demand `Bot` is not sufficient, since

$$\begin{aligned}\texttt{append}\ xs\ ys &\in \texttt{Bot} \rightarrow_{\text{CADE}} \langle[\texttt{Bot},\texttt{Top}],[\texttt{[]},\texttt{Bot}]\rangle\\ \texttt{!append}\ xs\ ys &\in \texttt{Bot} \rightarrow_{\text{CADE}} \langle[\texttt{Bot},\texttt{Top}],[\texttt{[]},\texttt{Bot}],\\ &\qquad\qquad [\texttt{Top}:\texttt{Bot},\texttt{Top}]\rangle\end{aligned}$$

We use abstract reduction [Sch94, Nöc93] to see

$$\texttt{append (Top : Bot) Top} \rightarrow^{\#} \texttt{Top : Bot}.$$

The demand $M \overset{\text{def}}{=} \langle \texttt{Bot}, \texttt{Top : Bot} \rangle$ is sufficient:

$$\begin{aligned}
\texttt{append}\ xs\ ys \in M &\to_{\text{CADE}} \langle [\texttt{Bot}, \texttt{Top}], [\texttt{[]}, \texttt{Bot}], \\
&\quad [\texttt{[]}, \texttt{Top : Bot}], [\texttt{[Top]}, \texttt{Bot}], \\
&\quad [\texttt{Top : Bot}, \texttt{Top}] \rangle \\
\texttt{!append}\ xs\ ys \in M &\to_{\text{CADE}} \langle [\texttt{Bot}, \texttt{Top}], [\texttt{[]}, \texttt{Bot}], \\
&\quad [\texttt{[]}, \texttt{Top : Bot}], [\texttt{[Top]}, \texttt{Bot}], \\
&\quad [\texttt{Top : Bot}, \texttt{Top}] \rangle
\end{aligned}$$

We can replace `append` by `!append` if the entire application's demand is at least $\langle \texttt{Bot}, \texttt{Top : Bot} \rangle$.

6.3.5 Absence

The analysis can also be used to analyze *absence* of an argument in a computation. We call an argument *absent in the computation* or *absent* for short if the result of the computation is constant for any value supplied as this argument [Bis97].

Definition 6.7. The argument of `f` is *absent*, iff $\forall x, y : \texttt{f}\ x \equiv_c \texttt{f}\ y$.

In order to analyze this property with our calculi, we need to define a demand expression $\texttt{Top}^+$ which stands for all Λ-expressions having a WHNF.

Definition 6.8. Let $\mathcal{A} = \{A_1, \ldots, A_m\}$

$$\texttt{Top}^+ = \langle \texttt{Fun}, \texttt{c}_{A_1,1} \underbrace{\texttt{Top} \ldots \texttt{Top}}_{\alpha(\texttt{c}_{A_1,1})}, \ldots, \texttt{c}_{A_m,|A_m|} \underbrace{\texttt{Top} \ldots \texttt{Top}}_{\alpha(\texttt{c}_{A_m,|A_m|})} \rangle.$$

Now we can distinguish two cases:

1. $\forall x : (\texttt{f}\ x)\Downarrow$

2. $\forall x : (\texttt{f}\ x)\Uparrow$

In order to analyze case 1, we formulate as input to the calculi: $\texttt{f}\ x \in \texttt{Top}^+$. If the analysis proves $\texttt{f}\ x \in \texttt{Top}^+ \to_{\text{ADE}} C$ where $\texttt{Bot} \leq_\gamma C$, case 1 applies. This is a consequence of monotonicity of `f` with respect to $\leq_c$. From theorem 3.67 we know that $\texttt{Bot} \leq_\gamma C$ is undecidable, yet in practice it is typically quite easy to see, since many results are of the form $\langle \texttt{Bot}, \ldots \rangle$.
In order to analyze case 2, we would like to input $\texttt{f}\ x \in \texttt{Bot}$ to the calculi and test if $\texttt{f}\ x \in \texttt{Bot} \to_{\text{ADE}} \texttt{Top}$. Since we will not simply obtain `Top`, but a demand C, we would have to test if C is equivalent to `Top`. $\texttt{Top} \leq_\gamma C$ is typically much more difficult to see than $\texttt{Bot} \leq_\gamma C$, because not many results $C \equiv \texttt{Top}$. We prefer a different approach: we analyze $\texttt{f}\ x \in \texttt{Top}^+$. If $\texttt{f}\ x \in \texttt{Top}^+ \to_{\text{CADE}} \emptyset$, then case 2 applies, since then $\theta \in \gamma(\emptyset) \iff \theta(\texttt{f}\ x) \in \gamma(\texttt{Top}^+)$.
The case 2 could also have been checked with a forward analysis, e.g. with abstract reduction [Sch94], if $\texttt{f Top} \to^{\#} \texttt{Bot}$ this case applies.

6.3.6 Neededness

A closed function f needs its argument if its value differs for some arguments, i.e. if the argument is not absent.

Definition 6.9. The argument of a function $f \in \Lambda^0$ is *needed* iff $\exists s, t \in \Lambda^0 : f\ s \not\equiv_c f\ t$.

Lemma 6.10. *Neededness implies strictness.*

Proof. Assume this does not hold, then there are $s, t \in \Lambda^0$ and a closed function f with $f\ s \not\equiv_c f\ t$ and $f\ \texttt{bot} \not\equiv_c \texttt{bot}$. Without loss of generality $\exists \mathbb{R}[\cdot] : \mathbb{R}[f\ s]\Downarrow \wedge \mathbb{R}[f\ t]\Uparrow$. For all $s \in \Lambda^0 : \texttt{bot} \leq_c s$

and thus f `bot` $\not\equiv_c$ `bot` requires f `bot` $\not\leq_c$ `bot`. On the other hand, monotonicity of contexts implies $\forall s : \mathbb{R}[f\ s] \not\leq_c \mathbb{R}[$`bot`$]$, but this contradicts the choice of f since $\mathbb{R}[$`bot`$] \equiv_c$ `bot`. □

Theorem 6.11. *A function $f \in \Lambda$ needs its argument iff f is strict and for some $s : f\ s \not\leq_c$* `bot`*.*

Proof.

$\Longrightarrow$: Definition 6.9 directly implies that $f\ s$ cannot diverge for all s and lemma 6.10 proved the implication of strictness.

$\Longleftarrow$: f's strictness implies f `bot` $\equiv_c$ `bot`, but there is an $s \in \Lambda : f\ s \not\equiv_c$ `bot`. `bot` and such an s satisfy definition 6.9. □

This theorem suggests how to analyze neededness. We analyze strictness of f in its argument and also $f\ x \in$ `Top`$^+$. If ADE obtains $C \not\equiv_\gamma \emptyset$ for the latter analysis, then there are bindings for x under which $f\ x$ does not diverge. While undecidable, there still are many practically relevant cases for which this is easily checked, e.g. $C \equiv \langle$`[]`$, \dots\rangle$ or $C \equiv \langle$`Fun`$, \dots\rangle$.
With respect to the evaluation of an expression the difference between strictness and neededness is that a strict argument in an application being evaluated can safely be evaluated whereas a needed argument will surely be evaluated.

7 Conclusion and future work

7.1 Conclusion

In chapter 3 we introduced demands which can specify particular sets of Λ-expressions. There we saw that demands are equally well suited for the specification of data values returned by programs as they are for the specification of non-terminating or undefined computations. Also, more complex structures such as infinite lists or Peano numbers can conveniently be specified by demands. We have investigated the expressive power of demands and have found it to be equivalent to that of Turing-machines and we have related concretizations of demands to fixpoints of functions. Furthermore, we investigated how the syntactic representation of some demands can be simplified without changing their concretization. Demand analysis and the two calculi ADE and CADE were treated in chapter 4. First a suitable data structure was introduced for the rules of the calculi to operate on. Then these rules were presented and each one was investigated with respect to its soundness and completeness. Finally, the soundness and completeness of the calculi as a whole was proved. Chapter 5 extended the base, laid out in chapters 2, 3 and 4, such that for example the same precision is available for demands specifying higher-order expressions as for those specifying saturated constructor applications and non-terminating expressions. Chapter 6 showed both examples motivating rules and those exhibiting the limitations of the calculi, and

presented different problems to which our calculi can be applied.

7.2 Future work

7.2.1 Local rules below `where`

Currently, the proof of soundness and completeness for applying a local rule below a `where`-expression needs to be done for every rule thus applied. Quite likely, a common criterion can be stated for the rules to allow this construction. This criterion would present some conditions a local rule needs to satisfy in order for its application below a `where`-expression to be sound and complete, respectively.

7.2.2 Heuristics

ADE and CADE were presented in chapter 4 as non-deterministic sets of rules. It would be interesting to investigate the effect of the choice of heuristic for rule application on space and time requirements.

7.2.3 Interplay of optimizations

While not strictly tied to demand analysis, the interplay of different optimizations would need further investigation, in particular, as an indication of the practical effect of optimizations. This would allow to measure the advantage of optimizations using semantic analyses over optimizations without such analyses. Consider the n-pack transformation from section 6.3.4. This transformation competes with deforestation, which, in its simplest formulation, requires a syntactic criterion, i.e. treelessness, for its application [Wad90]. Deforestation competes with the n-pack transformation since intermediate data structures that are already removed by deforestation can no longer be optimized by the n-pack transformation. For a statement relating the practical effectiveness of optimizations, and those using semantic analyses in particular, it is therefore insufficient to consider the optimizations separately, instead their cumulative effect needs investigation. This would also allow judging the overlap between competing optimizations. Competition is only one side of the “interplay” coin, the other is cooperation, i.e. to what extent does the application of one optimization introduce additional opportunities for the application of another. Further work in this direction will allow to better judge the contribution of the semantic analyses presented in this dissertation to optimization from a practical perspective.

7.2.4 Compiler integration

The prototypical implementation of the calculi is currently geared towards interactive use, and not towards automatic use as would be required should the analysis be integrated into a compiler. There are several issues involved: sensible resource bounds will need to be imposed on the individual analysis or on the entire analysis phase. Obvious candidates for such resources would be running time or tableau size, but others might be better suited. An automated choice is needed to determine which analyses might be reused. In the prototype this is the user’s choice: an analysis may be performed and its result marked reusable and such results may be stored and loaded to be available across sessions. Simplification of demands will likely become more important as complexity of the analyses increases, so experience is needed to identify the situations in which individual simplifications are beneficial.

Bibliography

[Abr90a] Samson Abramsky. Finding fixpoints in abstract interpretation. Technical Report DOC 90/20, Dept of Computing, Imperial College, 1990.

[Abr90b] Samson Abramsky. The lazy lambda calculus. In D. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.

[AH87] Samson Abramsky and Chris L. Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.

[AMJ94] Samson Abramsky, Pasquale Malacaria, and Radha Jagadeesan. Full abstraction for PCF. *Lecture Notes in Computer Science*, 789:1–??, 1994.

[Aug98] Lennart Augustsson. Cayenne – a language with dependent types. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 239–250, 1998.

[AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the 1993 Conference on Functional Programming Languages and Computer Architecture*, pages 31–41, Copenhagen, Denmark, June 1993.

[AWL94] Alexander Aiken, Edward L. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *Proceedings of the Twenty-First Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994.

[Bab87] Robert L. Baber. *The Spine of Software.* John Wiley and Sons, 1987.

[Bab97] Robert L. Baber. The Ariane 5 explosion as seen by a software engineer. available from the author, 1997.

[Bar84] Henk P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics.* North-Holland, Amsterdam, New York, 1984.

[BHA85] G. L. Burn, Chris L. Hankin, and Samson Abramsky. The theory for strictness analysis for higher order functions. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Structures*, number 217 in Lecture Notes in Computer Science, pages 42–62. Springer, 1985.

[Bis97] Sandip K. Biswas. A demand-driven set-based analysis. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, Paris, France, January 1997. ACM Press.

[Bur91] Geoffrey Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation.* Pitman, London, 1991.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 252–252. ACM Press, 1977.

[CC94] Patrick Cousot and Radhia Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and per analysis of functional languages). In *Proceedings of the 1994 International Conference on Computer Languages, Toulouse, France*, pages 95–112, Los Alamitos, California, USA, May 1994. IEEE Computer Society Press. invited paper.

[Dij76] Edsger W. Dijkstra. *A Dicipline of Programming.* Prentice Hall, 1976.

[DP90] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, Cambridge, 1990.

[DW90] Kei Davis and Philip Wadler. Strictness analysis in 4d. In *Glasgow functional programming workshop*, Aug 90.

[FB94] Sigbjørn Finne and Geoffrey L. Burn. Assessing the evaluation transformer model of reduction on the Spineless G-machine. Technical report, Imperial College of Science, Technology and Medicine, Department of Computing, 1994.

[FF95] Cormac Flanagan and Matthias Felleisen. Set-based analysis for full scheme and its use in soft-typing. Technical Report 95-253, Rice University, October 1995.

[FFK87] M. Felleisen, D. Friedman, and E. Kohlbecker. A syntactic theory of sequential control. *Theoretical Computer Science*, (52):205–237, 1987.

[GdW94] John P. Gallagher and D. Andre de Waal. Fast and precise regular approximation of logic programs. In Pascal Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 599–613, Santa Marherita, Ligure, Italy, June 1994. The MIT Press.

[Gie95] J. Giesl. Termination analysis for functional programs using term orderings. In Alan Mycroft, editor, *Static Analysis Symposium '95*, number 984 in Lecture Notes in Computer Science, pages 154–171. Springer, 1995.

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

[GS96] Robert Glück and Heine Sørensen. A roadmap to metacomputation and supercompilation. In *Partial Evaluation.* Springer Verlag, 1996.

[HJ92] Nevin Heintze and Joxan Jaffar. Semantic types for logic programs. In Frank Pfenning, editor, *Types in Logic Programming*, pages 141–155. The MIT Press, 1992.

[HJ94] Nevin Heintze and Joxan Jaffar. Set constraints and set-based analysis. In *Second Workshop on the Principles and Practice of Constraint Programming*, LNCS, pages 281–298, Orcas Island, Washington, May 1994. Springer.

[HL92] John Hughes and John Launchbury. Reversing abstract interpretations. In *Proceedings of the European Symposium on Programming*, number 582 in LNCS, Rennes, 1992. Springer-Verlag.

[Hoa69] C.A.R. (Tony) Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[Hug88] John Hughes. Backwards analysis of functional programs. In D. Bjørner, A. P. Ershov, and N. D. Jones, editors, *Partial Evaluation and Mixed Computation.* Elsevier Science Publishers, 1988.

[HW87] John Hughes and Philip Wadler. Projections for strictness analysis. In *Functional Programming Languages and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 385–407. Springer, 1987.

[JM79] N. Jones and S. Muchnick. Flow analysis and optimization of lisp-like structures. In S. Muchnick and

N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 102–131. Prentice Hall, 1979.

[Klo97] Norbert Klose. Implementierung der n-pack transformation im rahmen eines front-ends für haskell. Master's thesis, J.W.Goethe Universität, Frankfurt, 1997.

[Kor] Jacob Kornerup. Quotes screaming for formal methods. `www.cs.utexas.edu/users/~kornerup/quotes.html`.

[Kut00] Arne Kutzner. *Ein nictdeterministischer call-by-need Lambda-Kalkül mit erratic Choice: Operationale Semantik, Programmtransformationen und Anwendungen.* PhD thesis, Johann Wolfgang Goethe – Universität, Frankfurt, Germany, April 2000.

[LLF$^+$96] Jacques-Louis Lions, Lennart Løbeck, Jean-Luc Fauquembergue, Gilles Kahn, Wolfgang Kubbat, Stefan Levedag, Leonardo Mazzini, Didier Merle, Colin O'Halloran, Mauro Balduccini, Yvan Choquer, Remy Hergott, Bernard Humbert, and Eric Lefort. ARIANE 5, Flight 501 Failure, Report by the Inquiry Board. Technical report, Independent Inquiry Board of the ESA and CNES, 1996.

[LPJ96] John Launchbury and Simon Peyton Jones. State in Haskell. *Journal of functional programming*, 1996.

[Mil77] Robin Milner. Fully abstract models of typed λ-calculi. *Theoretical Computer Science*, 4(1):1–22, 1977.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *J.Comp.Sys.Sci*, 17:348–375, 1978.

[Mis84] Prateek Mishra. Towards a theory of types in prolog. In *Proceedings of the 1984 International Symposium on Logic Programming*, pages 289–298, Atlantic City, New Jersey, February 1984. IEEE-CS.

[Mit96] John C Mitchell. *Foundations for Programming Languages.* MIT Press, Cambridge, Mass., 1996.

[MS99] Andrew Moran and David Sands. Improvement in a lazy context: An operational theory for call-by-need. In *Proceedings of the Twenty-Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Antonio, TX, 1999. ACM Press. Extended version.

[MST96] Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. From operational semantics to domain theory. *Information and Computation*, (128):24–47, 1996.

[Myc81] Alan Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* PhD thesis, University of Edinburgh, 1981.

[Nöc93] Eric Nöcker. Strictness analysis using abstract reduction. In *Functional Programming Languages and Computer Architecture*, pages 255–265. ACM Press, 1993.

[Ong95] C.-H. L. Ong. Correspondence between operational and denotational semantics. In S. Abramsky, D. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science, Vol 4*, pages 269–356. Oxford University Press, 1995.

[Pan97] S. E. Panitz. *Generierung statischer Programminformation zur Kompilierung verzögert ausgewerteter funktionaler Programmiersprachen.* PhD thesis, FB Informatik, J.W. Goethe Universität Frankfurt, 1997. in German.

[Pap00] Dirk Pape. *Striktheitsanalysen funktionaler Sprachen.* PhD thesis, Fachbereich Mathematik und Informatik der Freien Universität Berlin, 2000.

[Pat96] Ross Paterson. Compiling laziness using projections. In *Static Analysis Symposium*, volume 1145 of *LNCS*, pages 255–269, Aachen, Germany, September 1996. Springer.

[PHA$^+$99] John Peterson [ed.], Kevin Hammond [ed.], Lennart Augustsson, Brian Boutel, Warren Burton, Joseph Fasel, Andrew D. Gordon, John Hughes, Paul Hudak, Thomas Johnsson, Mark Jones, Erik Meijer, Simon Peyton Jones, Alastair Reid, and Philip Wadler. Report on the programming language Haskell 98: A non-strict, purely functional language, 1999.

[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.

[Reh99] Dirk Rehberger. Erweiterung der Kontextanalyse für nicht-strikte funktionale Programmiersprachen um

Funktionskontexte. Master's thesis, Johann Wolfgang Goethe-Universität, Frankfurt, 1999.

[San96] David Sands. Proving the correctness of recursion-based automatic program transformations. *Theoretical Computer Science*, (167), 1996.

[San98a] D. Sands. Improvement theory and its applications. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 275–306. Cambridge University Press, 1998.

[San98b] David Sands. Computing with contexts, A simple approach. *Electronic Notes in Theoretical Computer Science*, 10, 1998. `http://www.elsevier.nl/locate/entcs/volume10.html`.

[Sch94] Marko Schütz. Striktheits-Analyse mittels abstrakter Reduktion für den Sprachkern einer nicht-strikten funktionalen Programmiersprache. Master's thesis, Johann Wolfgang Goethe-Universität, Frankfurt, 1994.

[Sla98] Gregory Slabodkin. Software glitches leave navy smart ship dead in the water. *Government Computer News*, 1998. `www.gcn.com/archives/gcn/1998/july13/cov2.htm`.

[Sør96] Morten Heine Sørensen. Turchin's supercompiler revisited. Master's thesis, University of Copenhagen, Department of Computer Science, DIKU, 1996.

[SS99] Manfred Schmidt-Schauß. Funktionale Programmierung I. Lecture notes in german, 1999.

[Tre94] Guy Tremblay. *Parallel Implementation of Lazy Functional Languages Using Abstract Demand Propagation*. PhD thesis, School of Computer Science, McGill University, 1994.

[Tur37] Alan M. Turing. The $\mathfrak{p}$-functions in $\lambda - k$-conversion. *J. Symbolic Logic*, 2:164–, 1937.

[Wad87] Phil Wadler. Strictness analysis on non-flat domains (by abstract interpretation over finite domains). In Samson Abramsky and Chris L. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 12. Ellis Horwood Limited, Chichester, 1987.

[Wad90] Phil Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[Wra85] S. Wray. A new strictness detection algorithm. In Lennart Augustsson et al., editor, *Proceedings of the Workshop on Implementation of Functional Languages*, Göteborg, Sweden, 1985. Programming Methodology Group, Department of Computer Science, Chalmers University of Technology.

Symbol Index

Index