Mode Analysis Domains for Typed Logic Programs

Jan–Georg Smaus[1], Patricia M. Hill[2], and Andy King[3]

[1] INRIA-Rocquencourt, France,
jan.smaus@inria.fr
[2] University of Leeds, United Kingdom
hill@scs.leeds.ac.uk
[3] University of Kent at Canterbury, United Kingdom
a.m.king@ukc.ac.uk

Abstract. Precise mode information is important for compiler optimisations and in program development tools. Within the framework of abstract compilation, the precision of a mode analysis depends, in part, on the expressiveness of the abstract domain and its associated abstraction function. This paper considers abstract domains for polymorphically typed logic programs and shows how specialised domains may be constructed for each type in the program. These domains capture the degree of instantiation to a high level of precision. By providing a generic definition of abstract unification, the abstraction of a program using these domains is formalised. The domain construction procedure is fully implemented using the Gödel language and tested on a number of example programs to demonstrate the viability of the approach.

Note: Some proofs have been omitted for space reasons. They can be found in the full version of this paper [17].

1 Introduction

1.1 Background

Typed logic programming languages such as Mercury [19] and Gödel [10] use a *prescriptive* type system [15], which restricts the underlying syntax so that only meaningful expressions are allowed. This enables most typographical errors and inconsistencies in the knowledge representation to be detected at compile time. An increasing number of applications using typed languages are being developed.

Our notion of *modes* is, in contrast, a *descriptive* one [3, 7]: Modes characterise the degree to which program variables are instantiated at certain program points. This information can be used to underpin optimisations such as the specialisation of unification and the removal of backtracking, and to support determinacy analysis [9]. When a mode analysis is formulated in terms of abstract interpretation, the program execution is traced using *descriptions* of data (the *abstract* domain) rather than *actual* data, and operations on these descriptions rather than operations on the actual data. The precision of a mode analysis depends, in part, on the expressiveness of the abstract domain.

1.2 Contribution

The main contribution of this paper is to describe a generic method of deriving precise abstract domains for mode analysis from the type declarations of a typed program. Each abstract domain is specialised for a particular type and characterises a set of possible modes for terms of that type. In particular it characterises the property of *termination*, well-known for lists as *nil*-termination.

The procedure for constructing such domains is implemented (in Gödel) for Gödel programs. By incorporating the constructed domains into a mode analyser, the viability of the approach is demonstrated.

The abstract domains are used in an *abstract compilation* [4] framework: A program is abstracted by replacing each unification with an abstract counterpart, and then the abstract program is evaluated by applying a standard operational semantics to it.

We believe that this work is the natural generalisation of [3, 5] and takes the idea presented there to its limits: Our abstract domains provide the highest degree of precision that a generic domain construction should provide. Not only can this work be used directly for the mode analysis of typed logic programs, but it could be used as a basis for constructing (more pragmatic) domains as well as providing a unifying theory for other proposals.

The paper is organised as follows. Section 2 introduces three examples. Section 3 defines some syntax. Section 4 defines the concepts for terms and types that are used in the definition of abstract domains. Section 5 defines abstract domains and programs, and the relationship between concrete and abstract programs. Section 6 reports on experiments. Section 7 concludes.

2 Motivating and Illustrative Examples

We introduce three examples that we use throughout the paper. The syntax is that of the typed language Gödel [10], to avoid any confusion with the (untyped) language Prolog. Variables and (type) parameters begin with lower case letters; other alphabetic symbols begin with upper case letters. We use `Integer` (abbreviated as `Int`) to illustrate a type containing only constants $(1, 2, 3 \dots)$.

Example 2.1. This is the usual list type. We give its declarations to illustrate the type description language of Gödel.

```
CONSTRUCTOR     List/1.
CONSTANT        Nil: List(u).
FUNCTION        Cons: u * List(u) -> List(u).
```

`List` is a (type) constructor; `u` is a type parameter; `Nil` is a constant of type `List(u)`; and `Cons` is the usual list constructor. We use the standard list notation $[\dots|\dots]$ where convenient. It is common to distinguish *nil-terminated* lists from *open* lists. For example, [] and $[\mathtt{1}, \mathtt{x}, \mathtt{y}]$ are nil-terminated, but $[\mathtt{1}, \mathtt{2}|\mathtt{y}]$ is open.

Previous approaches cannot deal with the following two examples [3, 5, 21].

Example 2.2. This example was invented to disprove a common point of criticism that "list flattening" cannot be realised in Gödel, that is terms such as $[1,[2,3]]$ cannot be defined, let alone flattened. The `Nests` module formalises nested lists by the type `Nest(v)`. A trivial nest is constructed using function `E`, a complex nest by "nesting" a list of nests using function `N`. The declaration for `N` is remarkable in that the range type, `Nest(v)`, is a proper sub"term" of the argument type `List(Nest(v))`.

```
IMPORT            Lists, Integers.
CONSTRUCTOR       Nest/1.
FUNCTION          E: v -> Nest(v);
                  N: List(Nest(v)) -> Nest(v).
```

Example 2.3. A table is a data structure containing an ordered collection of nodes, each of which has two components, a key (of type `String`) and a value, of arbitrary type. We give part of the `Tables` module which is provided as a system module in Gödel.

```
IMPORT       Strings.
BASE         Balance.
CONSTRUCTOR  Table/1.
CONSTANT     Null: Table(u);
             LH, RH, EQ: Balance.
FUNCTION     Node: Table(u) * String * u * Balance * Table(u) -> Table(u).
```

`Tables` is implemented in Gödel as an AVL-tree [22]: A non-leaf node has a *key* argument, a *value* argument, arguments for the left and right subtrees, and an argument which represents balancing information.

3 Notation and Terminology

The set of polymorphic types is given by the term structure $T(\Sigma_\tau, U)$ where Σ_τ is a finite alphabet of **constructor** symbols which includes at least one **base** (constructor of arity 0), and U is a countably infinite set of **parameters** (type variables). We define the order $\prec$ on types as the order induced by some (for example lexicographical) order on constructor and parameter symbols, where parameter symbols come before constructor symbols. Parameters are denoted by u, v. A tuple of *distinct* parameters ordered with respect to $\prec$ is denoted by $\bar{u}$. Types are denoted by $\sigma, \rho, \tau, \phi, \omega$ and tuples of types by $\bar{\sigma}, \bar{\tau}$.

Let Σ_f be an alphabet of **function** (term constructor) symbols which includes at least one **constant** (function of arity 0) and let Σ_p be an alphabet of predicate symbols. Each symbol in Σ_f (resp. Σ_p) has its *type* as subscript. If $f_{\langle \tau_1 \ldots \tau_n, \tau \rangle} \in \Sigma_f$ (resp. $p_{\langle \tau_1 \ldots \tau_n \rangle} \in \Sigma_p$) then $\langle \tau_1, \ldots, \tau_n \rangle \in T(\Sigma_\tau, U)^\star$ and $\tau \in T(\Sigma_\tau, U) \setminus U$. If $f_{\langle \tau_1 \ldots \tau_n, \tau \rangle} \in \Sigma_f$, then every parameter occurring in $\langle \tau_1, \ldots, \tau_n \rangle$ must also occur in τ. This condition is called **transparency condition**. We call τ the **range type** of $f_{\langle \tau_1 \ldots \tau_n, \tau \rangle}$ and $\{\tau_1 \ldots \tau_n\}$ its **domain types**. A symbol is often written without its type if it is clear from the context. Terms and atoms are defined in the usual way [10, 16]. In this terminology, if a term *has* a type σ, it also *has* every *instance* of σ.[1] If V is a countably infinite set of vari-

[1] For example, the term `Nil` has type `List(u)`, `List(Int)`, `List(Nest(Int))` etc.

ables, then the triple $L = \langle \Sigma_p, \Sigma_f, V \rangle$ defines a **polymorphic many-sorted first order language** over $T(\Sigma_\tau, U)$. Variables are denoted by x, y; terms by t, r, s; tuples of *distinct* variables by $\bar{x}, \bar{y}$; and a tuple of terms by $\bar{t}$. The set of variables in a syntactic object o is denoted by $vars(o)$.

Programs are assumed to be in **normal form.** Thus a **literal** is an equation of the form $x =_{\langle u,u \rangle} y$ or $x =_{\langle u,u \rangle} f(\bar{y})$, where $f \in \Sigma_f$, or an atom $p(\bar{y})$, where $p \in \Sigma_p$. A **query** G is a conjunction of literals. A clause is a formula of the form $p(\bar{y}) \leftarrow G$. If S is a set of clauses, then the tuple $P = \langle L, S \rangle$ defines a **polymorphic many-sorted logic program**.

A **substitution** (denoted by Θ) is a mapping from variables to terms which is the identity almost everywhere. The **domain** of a substitution Θ is $dom(\Theta) = \{x \mid x\Theta \neq x\}$. The application of a substitution Θ to a term t is denoted as $t\Theta$. **Type substitutions** are defined analogously and denoted by Ψ.

4 The Structure of Terms and Types

An *abstract* term characterises the structure of a concrete term. It is clearly a crucial choice in the design of abstract domains *which* aspects of the concrete structure should be characterised [21, 23]. In this paper we show how this choice can be based naturally on the information contained in the type subscripts of the function symbols in Σ_f. This information is formalised in this section. First we formalise the relationship between the range type of a function to its domain types. We then define *termination* of a term, as well as functions which extract certain subterms of a term. In the following, we assume a fixed polymorphic many-sorted first order language $L = \langle \Sigma_p, \Sigma_f, V \rangle$ over $T(\Sigma_\tau, U)$.

4.1 Relations between Types

Definition 4.1 (subterm type). A type σ is a **direct subterm type of** ϕ (denoted as $\sigma \lhd \phi$) if there is $f_{\langle \tau_1 \ldots \tau_n, \tau \rangle} \in \Sigma_f$ and a type substitution Ψ such that $\tau\Psi = \phi$ and $\tau_i\Psi = \sigma$ for some $i \in \{1, \ldots, n\}$. The transitive, reflexive closure of $\lhd$ is denoted as $\lhd^*$. If $\sigma \lhd^* \phi$, then σ is a **subterm type of** ϕ.

The relation $\lhd$ can be visualised as a *type graph* (similarly defined in [18, 23]). The type graph for a type ϕ is a directed graph whose nodes are subterm types of ϕ. The node ϕ is called the *initial node.* There is an edge from σ_1 to σ_2 if and only if $\sigma_2 \lhd \sigma_1$.

Example 4.1. Figure 1 shows a type graph for each example in Sect. 2. The left hand type graph illustrates Ex. 2.1 where $\mathtt{u} \lhd \mathtt{List(u)}$ and $\mathtt{List(u)} \lhd \mathtt{List(u)}$. The other two type graphs illustrate Exs. 2.2 and 2.3, respectively.

A **simple type** is a type of the form $C(\bar{u})$, where $C \in \Sigma_\tau$. We impose the following two restrictions on the language.

Simple Range Condition: For all $f_{\langle \tau_1 \ldots \tau_n, \tau \rangle} \in \Sigma_f$, τ is a simple type.

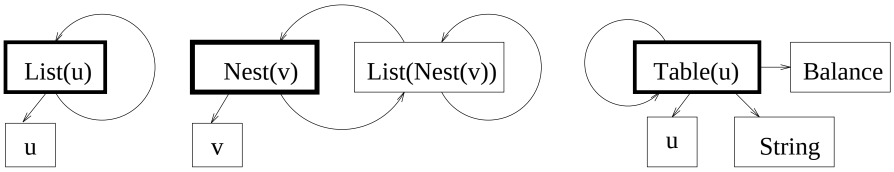

Fig. 1. Some type graphs, with initial node highlighted

Reflexive Condition: For all $C \in \Sigma_\tau$ and types $\sigma = C(\bar{\sigma}), \tau = C(\bar{\tau})$, if $\sigma \lhd^* \tau$, then σ is a sub"term" (in the syntactic sense) of τ.

The Simple Range Condition allows for the construction of an abstract domain for a type such as $\mathtt{List}(\sigma)$ to be described independently of the type σ. In Mercury (and also in typed *functional* languages such as ML or Haskell), this condition is enforced by the syntax [19]. Being able to violate this condition can be regarded as an artefact of the Gödel syntax.

The Reflexive Condition ensures that, for a program and a given query, there are only finitely many types and hence, the abstract program has only finitely many abstract domains and the type graphs are always finite. It rules out, for example, a function symbol of the form $f_{\langle List(Int), List(u)\rangle}$ since this would imply that $\mathtt{List(Int)} \lhd^* \mathtt{List(u)}$. We do not know of any real programs that violate the Reflexive Condition or the Simple Range Condition.

Definition 4.2 (recursive type and non-recursive subterm type). A type σ is a **recursive type of** ϕ (denoted as $\sigma \bowtie \phi$) if $\sigma \lhd^* \phi$ and $\phi \lhd^* \sigma$.

A type σ is a **non-recursive subterm type (NRS) of** ϕ if $\phi \not\lhd^* \sigma$ and there is a type τ such that $\sigma \lhd \tau$ and $\tau \bowtie \phi$. We write $\mathcal{N}(\phi) = \{\sigma \mid \sigma \text{ is an NRS of } \phi\}$. If $\mathcal{N}(\phi) = \{\sigma_1, \ldots, \sigma_m\}$ and $\sigma_j \prec \sigma_{j+1}$ for all $j \in \{1, \ldots, m-1\}$, we abuse notation and denote the *tuple* $\langle \sigma_1, \ldots, \sigma_m \rangle$ by $\mathcal{N}(\phi)$ as well.

It follows immediately from the definition that, for any types ϕ, σ, we have $\phi \bowtie \phi$ and, if $\sigma \in \mathcal{N}(\phi)$, then $\sigma \not\bowtie \phi$. Consider the type graph for ϕ. The recursive types of ϕ are all the types in the strongly connected component (SCC) containing ϕ. The non-recursive subterm types of ϕ are all the types σ not in the SCC but such that there is an edge from the SCC containing ϕ to σ.

Example 4.2. Consider again Ex. 4.1 and Fig. 1. Then $\mathtt{List(u)} \bowtie \mathtt{List(u)}$, and this is non-trivial in that, in the type graph for `List(u)`, there is an edge from $\mathtt{List(u)}$ to itself. Furthermore $\mathtt{List(Nest(v))} \bowtie \mathtt{Nest(v)}$. Non-recursive subterm types of simple types are often parameters, as in $\mathcal{N}(\mathtt{List(u)}) = \langle \mathtt{u} \rangle$ and $\mathcal{N}(\mathtt{Nest(v)}) = \langle \mathtt{v} \rangle$. However, this is not always the case, since $\mathcal{N}(\mathtt{Table(u)}) = \langle \mathtt{u}, \mathtt{Balance}, \mathtt{String} \rangle$.

The following simple lemma is used in the proof of Lemma 4.2.

Lemma 4.1. Let ϕ, τ, σ be types so that $\sigma \lhd^* \tau \lhd^* \phi$ and $\sigma \bowtie \phi$. Then $\tau \bowtie \phi$.

Proof. Since $\sigma \bowtie \phi$, it follows that $\phi \lhd^* \sigma$. Thus, since $\sigma \lhd^* \tau$, it follows that $\phi \lhd^* \tau$. Furthermore $\tau \lhd^* \phi$, and therefore $\tau \bowtie \phi$. □

4.2 Traversing Concrete Terms

From now on, we shall often annotate a term t with a type ϕ by writing t^ϕ. The use of this notation *always* implies that the type of t must be an *instance* of ϕ. The annotation ϕ gives the (type) context in which t is used. If S is a set of terms, then S^ϕ denotes the set of terms in S, each annotated with ϕ.

Definition 4.3 (subterm). Let t^ϕ be a term where $t = f_{\langle \tau_1 \dots \tau_n, \tau \rangle}(t_1, \dots, t_n)$ and $\phi = \tau\Psi$. Then $t_i^{\tau_i \Psi}$ is a **subterm of** t^ϕ (denoted as $t_i^{\tau_i \Psi} \lhd t^\phi$) for each $i \in \{1, \dots, n\}$. As in Def. 4.1, the transitive, reflexive closure of $\lhd$ is denoted by $\lhd^*$.

It can be seen that $s^\sigma \lhd^* t^\phi$ implies $\sigma \lhd^* \phi$. When the superscripts are ignored, the above is the usual definition of a subterm. The superscripts provide a uniform way of describing the "polymorphic type relationship" between a term and its subterms, which is independent of further instantiation.

Example 4.3. $\mathtt{x}^{\mathtt{v}}$ is a subterm of $\mathtt{E(x)}^{\mathtt{Nest(v)}}$, and $\mathtt{7}^{\mathtt{v}}$ is a subterm of $\mathtt{E(7)}^{\mathtt{Nest(v)}}$.

Definition 4.4 (recursive subterm). Let s^σ and t^τ be terms such that $s^\sigma \lhd^* t^\tau$, and ϕ a type such that $\sigma \bowtie \phi$ and $\tau \lhd^* \phi$. Then s^σ is a **ϕ-recursive subterm of** t^τ. If furthermore $\tau = \phi$, then s^σ is a **recursive subterm of** t^τ.

In particular, for every type ϕ, a variable is always a ϕ-recursive subterm of itself. The correspondence between subterms and subterm types can be illustrated by drawing the term as tree that resembles the corresponding type graph.

Example 4.4.
The term tree for $t = \mathtt{N([E(7)])}^{\mathtt{Nest(v)}}$ is given in Fig. 2 where the node for t is highlighted. Each box drawn with solid lines stands for a subterm. We can map this tree onto the type graph for $\mathtt{Nest(v)}$ in Fig. 1 by replacing the subgraphs enclosed with dotted lines with corresponding nodes in the type graph. Thus the recursive subterms of t occur in the boxes corresponding to nodes in the SCC of $\mathtt{Nest(v)}$. All subterms of t except $\mathtt{7}^{\mathtt{v}}$ are recursive subterms of t.

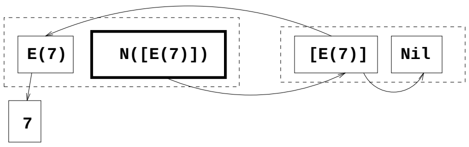

Fig. 2: Term tree for $\mathtt{N([E(7)])}^{\mathtt{Nest(v)}}$

Note that $\mathtt{E(7)}^{\mathtt{Nest(v)}}$ is a $\mathtt{Nest(v)}$-recursive subterm of $\mathtt{[E(7)]}^{\mathtt{List(Nest(v))}}$ (in Def. 4.4, take $\sigma = \phi = \mathtt{Nest(v)}$ and $\tau = \mathtt{List(Nest(v))}$). However, $\mathtt{E(7)}^{\mathtt{u}}$ is not a recursive subterm of $\mathtt{[E(7)]}^{\mathtt{List(u)}}$. Thus whether or not a member of a list should be regarded as a recursive subterm of that list depends on the context.

We now define *termination* of a term. For a term t^ϕ, where ϕ is simple, termination means that no recursive subterm of t^ϕ is a variable.

Definition 4.5 (termination function $\mathcal{Z}$). Let t^τ be a term and ϕ be a type such that $\tau \bowtie \phi$. Define $\mathcal{Z}(t^\tau, \phi) = \mathit{false}$ if a ϕ-recursive subterm of t^τ is a variable, and *true* otherwise. For a set S^τ of terms define $\mathcal{Z}(S^\tau, \phi) = \bigwedge_{t\in S} \mathcal{Z}(t^\tau, \phi)$. We omit τ in the expression $\mathcal{Z}(t^\tau, \phi)$ whenever $\phi = \tau$. We say that t is **terminated** if τ is simple and $\mathcal{Z}(t, \tau) = \mathit{true}$, and t is **open** if it is not terminated.

Example 4.5. Any variable x is open. The term 7 has no variable subterm, so $\mathcal{Z}(7, \mathtt{Int}) = \mathit{true}$ and 7 is terminated. The term $[\mathtt{x}]^{\mathtt{List(u)}}$ has itself and $\mathtt{Nil}^{\mathtt{List(u)}}$ as recursive subterms, so $\mathcal{Z}([\mathtt{x}], \mathtt{List(u)}) = \mathit{true}$ and $[\mathtt{x}]$ is terminated. However, $[\mathtt{x}]^{\mathtt{List(Nest(v))}}$ has $\mathtt{x}^{\mathtt{Nest(v)}}$ as a Nest(v)-recursive subterm, so $\mathcal{Z}([\mathtt{x}]^{\mathtt{List(Nest(v))}}, \mathtt{Nest(v)}) = \mathit{false}$. Furthermore, $\mathtt{N}([\mathtt{x}])^{\mathtt{Nest(v)}}$ has $\mathtt{x}^{\mathtt{Nest(v)}}$ as a recursive subterm, so $\mathcal{Z}(\mathtt{N}([\mathtt{x}]), \mathtt{Nest(v)}) = \mathit{false}$ and $\mathtt{N}([\mathtt{x}])$ is open.

The abstract domain should also characterise the instantiation of subterms of a term. We define functions which extract sets of subterms from a term.

Definition 4.6 (extractor $\mathcal{E}^\sigma$ for σ). Let t^τ be a term and ϕ, σ be types such that $\tau \bowtie \phi$ and $\sigma \in \mathcal{N}(\phi)$. Let R be the set of ϕ-recursive subterms of t^τ. Define

$$\mathcal{E}^\sigma(t^\tau, \phi) = \mathit{vars}(R) \,\cup\, \{s \mid r^\rho \in R \text{ and } s^\sigma \lhd r^\rho\}.$$

For a set S^τ of terms define $\mathcal{E}^\sigma(S^\tau, \phi) = \bigcup_{t\in S} \mathcal{E}^\sigma(t^\tau, \phi)$. As with $\mathcal{Z}$, we write $\mathcal{E}^\sigma(t^\tau, \tau)$ simply as $\mathcal{E}^\sigma(t, \tau)$.

Example 4.6. For $\mathtt{N}([\mathtt{E}(7)])$ of type Nest(Int), we have $\mathcal{E}^{\mathtt{v}}(\mathtt{N}([\mathtt{E}(7)]), \mathtt{Nest(v)}) = \{7\}$. The type Table(u) has three non-recursive subterm types u, Balance and String, and so there are three extractor functions: $\mathcal{E}^{\mathtt{u}}$, which extracts all value subterms; $\mathcal{E}^{\mathtt{Balance}}$, which extracts the argument containing balancing information; and $\mathcal{E}^{\mathtt{String}}$, which extracts all key subterms. Note that for a term t of type Table(String), both $\mathcal{E}^{\mathtt{String}}(t)$ and $\mathcal{E}^{\mathtt{u}}(t)$ would contain terms of type String.

Note that a priori, the extracted terms have no type annotation. This is because, in the proofs, we sometimes need to write an expression such as $\mathcal{E}^\sigma(\mathcal{E}^\rho(t, \tau)^{\rho\Psi}, \phi)$, which reads: first compute $\mathcal{E}^\rho(t, \tau)$, then annotate it with $\rho\Psi$, then pass it to $\mathcal{E}^\sigma$.

Note also that if t has a ϕ-recursive subterm which is a variable, then this variable is always extracted. Intuitively this is because this variable might later be instantiated to a term which has variable subterms of type σ. Thus the property "$\mathcal{E}^\sigma(t, \tau)$ does not contain variables" is closed under instantiation.

The following theorem shows that $\mathcal{Z}$ and $\mathcal{E}^\sigma$ can be expressed in terms of the immediate subterms of a term. This provides the basis for defining the abstraction of a (normal form) equation in a concrete program, which naturally involves a term and its immediate subterms.

Theorem 4.2. Let $t = f_{\langle \tau_1 \ldots \tau_n, \tau\rangle}(t_1, \ldots, t_n)$ be a term and $\sigma \in \mathcal{N}(\tau)$. Then

$$\begin{aligned}\mathcal{Z}(t, \tau) &= \bigwedge_{\tau_i \bowtie \tau} \mathcal{Z}(t_i^{\tau_i}, \tau)\\ \mathcal{E}^\sigma(t, \tau) &= \{t_i \mid \tau_i = \sigma\} \,\cup\, \bigcup_{\tau_i \bowtie \tau} \mathcal{E}^\sigma(t_i^{\tau_i}, \tau).\end{aligned}$$

Proof. If for some $i \in \{1, \ldots, n\}$ where $\tau_i \bowtie \tau$, r^ρ is a τ-recursive subterm of $t_i^{\tau_i}$, then $\rho \bowtie \tau$ and $r^\rho \lhd^* t^\tau$. Thus r^ρ is a τ-recursive subterm of t^τ.

If r^ρ is a τ-recursive subterm of t^τ, then either $r^\rho = t^\tau$ or, for some $i \in \{1, \ldots, n\}$, $r^\rho \lhd^* t_i^{\tau_i}$. In the latter case, $\rho \lhd^* \tau_i$, $\tau_i \lhd \tau$ and $\rho \bowtie \tau$. Hence, by Lemma 4.1, $\tau_i \bowtie \tau$ so that r^ρ is a τ-recursive subterm of $t_i^{\tau_i}$.

Thus the τ-recursive subterms of t are t, together with the τ-recursive subterms of $t_i^{\tau_i}$, where $\tau_i \bowtie \tau$. The result then follows from Defs. 4.5 and 4.6. □

Consider simple types ϕ, τ such that $\tau\Psi \bowtie \phi$ for some type substitution Ψ (for example $\phi = \mathtt{Nest(v)}$, $\tau = \mathtt{List(u)}$ and and $\Psi = \{\mathtt{u/Nest(v)}\}$). The following theorem relates ϕ with τ with respect to the termination and extractor functions.

Theorem 4.3 (Proof see [17]). Let ϕ and τ be simple types such that $\tau\Psi \bowtie \phi$ for some Ψ, let t be a term having a type which is an instance of $\tau\Psi$, and $\sigma \in \mathcal{N}(\phi)$. Then

$$\mathcal{Z}(t^{\tau\Psi}, \phi) = \qquad \mathcal{Z}(t, \tau) \wedge \bigwedge_{\substack{\rho \in \mathcal{N}(\tau) \\ \rho\Psi \bowtie \phi}} \mathcal{Z}(\mathcal{E}^\rho(t,\tau)^{\rho\Psi}, \phi) \tag{1}$$

$$\mathcal{E}^\sigma(t^{\tau\Psi}, \phi) = \bigcup_{\substack{\rho \in \mathcal{N}(\tau) \\ \rho\Psi = \sigma}} \mathcal{E}^\rho(t, \tau) \cup \bigcup_{\substack{\rho \in \mathcal{N}(\tau) \\ \rho\Psi \bowtie \phi}} \mathcal{E}^\sigma(\mathcal{E}^\rho(t,\tau)^{\rho\Psi}, \phi) \tag{2}$$

Example 4.7. First let $\phi = \tau = \mathtt{List(u)}$ and Ψ be the identity. Then by Def. 4.2 there is no ρ such that $\rho \in \mathcal{N}(\tau)$ and $\rho\Psi \bowtie \phi$. Therefore in both equations of Thm. 4.3, the right half of the right hand side is empty. Furthermore there is exactly one ρ such that $\rho\Psi = \sigma$, namely $\rho = \sigma$. Thus the equations read

$$\mathcal{Z}(t, \tau) = \mathcal{Z}(t, \tau) \tag{1}$$

$$\mathcal{E}^\sigma(t, \tau) = \mathcal{E}^\sigma(t, \tau) \tag{2}$$

Similarly, Thm. 4.3 reduces to a trivial statement for Ex. 2.3 and in fact for most types that are commonly used. However for Ex. 4.4, Thm. 4.3 says that

$$\begin{aligned} &\mathcal{Z}([\mathtt{E(7)}]^{\mathtt{List(Nest(v))}}, \mathtt{Nest(v)}) = \\ &\qquad \mathcal{Z}([\mathtt{E(7)}], \mathtt{List(u)}) \wedge \mathcal{Z}(\mathcal{E}^{\mathtt{u}}([\mathtt{E(7)}], \mathtt{List(u)}), \mathtt{Nest(v)}) \end{aligned} \tag{1}$$

$$\begin{aligned} &\mathcal{E}^{\mathtt{v}}([\mathtt{E(7)}]^{\mathtt{List(Nest(v))}}, \mathtt{Nest(v)}) = \\ &\qquad \emptyset \qquad \cup\, \mathcal{E}^{\mathtt{v}}(\mathcal{E}^{\mathtt{u}}([\mathtt{E(7)}], \mathtt{List(u)}), \mathtt{Nest(v)}) \end{aligned} \tag{2}$$

5 Abstract Terms and Abstract Programs

In this section, we first define the abstraction function for terms. Then we define termination and extractor functions for abstract terms. Finally, we define an abstract program and show how it approximates its concrete counterpart.

5.1 Abstraction of Terms

We first define an abstract domain for each type. Each abstract domain is a term structure, built using the constant symbols $\texttt{Bot}$, $\texttt{Any}$, $\texttt{Ter}$, $\texttt{Open}$, and the function symbols $C^{\mathcal{A}}$, for each $C \in \Sigma_\tau$.

Definition 5.1 (abstract domain). If ϕ is a parameter, define

$$\mathcal{D}_\phi = \{\texttt{Bot}, \texttt{Any}\}.$$

If $C(\bar{u})$ is a simple type with $\mathcal{N}(C(\bar{u})) = \langle \sigma_1, \ldots, \sigma_m \rangle$ and $\phi = C(\bar{u})\Psi$ where Ψ is a type substitution, define

$$\mathcal{D}_\phi = \{C^{\mathcal{A}}(b_1, \ldots, b_m, \texttt{Ter}) \mid b_j \in \mathcal{D}_{\sigma_j \Psi}\} \cup \{C^{\mathcal{A}}(\underbrace{\texttt{Any}, \ldots, \texttt{Any}}_{m \text{ times}}, \texttt{Open}),\ \texttt{Any}\}.$$

$\mathcal{D}_\phi$ is the **abstract domain for** ϕ. If $b \in \mathcal{D}_\phi$, then b is an **abstract term for** ϕ.

In [17], it is proven that every domain is well-defined. We shall see later that if an abstract term $C^{\mathcal{A}}(b_1, \ldots, b_m, \texttt{Ter})$ abstracts a term t, then each b_j corresponds to a non-recursive subterm type σ_j of $C(\bar{u})$. It characterises the degree of instantiation of the subterms extracted by $\mathcal{E}^{\sigma_j}$.

The termination flags $\texttt{Ter}$ and $\texttt{Open}$ in the last argument position of an abstract term are Boolean flags. The flag $\texttt{Ter}$ abstracts the property of a term being terminated and $\texttt{Open}$ that of being open. Note that for some types, for example $\texttt{Int}$, a term can be open only if it is a variable. In these cases, the termination flag can be omitted in the implementation (see Sect. 6).

Example 5.1. Consider the examples in Sect. 2 and Fig. 1.

$$\mathcal{D}_{\texttt{Int}} = \{\texttt{Int}^{\mathcal{A}}(\texttt{Ter}),\ \texttt{Int}^{\mathcal{A}}(\texttt{Open}),\ \texttt{Any}\}.$$

The following examples illustrate that Def. 5.1 is "parametric".

$$\begin{aligned}
\mathcal{D}_{\texttt{List(Int)}} &= \{\texttt{List}^{\mathcal{A}}(i, \texttt{Ter}) \mid i \in \mathcal{D}_{\texttt{Int}}\} \cup \{\texttt{List}^{\mathcal{A}}(\texttt{Any}, \texttt{Open}),\ \texttt{Any}\}\\
\mathcal{D}_{\texttt{List(String)}} &= \{\texttt{List}^{\mathcal{A}}(i, \texttt{Ter}) \mid i \in \mathcal{D}_{\texttt{String}}\} \cup \{\texttt{List}^{\mathcal{A}}(\texttt{Any}, \texttt{Open}),\ \texttt{Any}\}\\
\mathcal{D}_{\texttt{List(u)}} &= \{\texttt{List}^{\mathcal{A}}(i, \texttt{Ter}) \mid i \in \mathcal{D}_{\texttt{u}}\} \cup \{\texttt{List}^{\mathcal{A}}(\texttt{Any}, \texttt{Open}),\ \texttt{Any}\}.
\end{aligned}$$

Some further examples are, assuming that $\texttt{u} \prec \texttt{Balance} \prec \texttt{String}$:

$$\begin{aligned}
\mathcal{D}_{\texttt{Balance}} &= \{\texttt{Balance}^{\mathcal{A}}(\texttt{Ter}),\ \texttt{Balance}^{\mathcal{A}}(\texttt{Open}),\ \texttt{Any}\}\\
\mathcal{D}_{\texttt{String}} &= \{\texttt{String}^{\mathcal{A}}(\texttt{Ter}),\ \texttt{String}^{\mathcal{A}}(\texttt{Open}),\ \texttt{Any}\}\\
\mathcal{D}_{\texttt{Table(Int)}} &= \{\texttt{Table}^{\mathcal{A}}(i, b, s, \texttt{Ter}) \mid i \in \mathcal{D}_{\texttt{Int}}, b \in \mathcal{D}_{\texttt{Balance}}, s \in \mathcal{D}_{\texttt{String}}\} \cup\\
&\quad\ \{\texttt{Table}^{\mathcal{A}}(\texttt{Any}, \texttt{Any}, \texttt{Any}, \texttt{Open}),\ \texttt{Any}\}\\
\mathcal{D}_{\texttt{Nest(Int)}} &= \{\texttt{Nest}^{\mathcal{A}}(i, \texttt{Ter}) \mid i \in \mathcal{D}_{\texttt{Int}}\} \cup \{\texttt{Nest}^{\mathcal{A}}(\texttt{Any}, \texttt{Open}),\ \texttt{Any}\}.
\end{aligned}$$

We now define an order on abstract terms which has the usual interpretation that "smaller" stands for "more precise".

Definition 5.2 (order $<$ on abstract terms). For the termination flags define $\texttt{Ter} < \texttt{Open}$. For abstract terms, $<$ is defined as follows:

$$\begin{aligned} \texttt{Bot} &< b && \text{if } b \neq \texttt{Bot}, \\ b &< \texttt{Any} && \text{if } b \neq \texttt{Any}, \\ C^{\mathcal{A}}(b_1, \ldots, b_m, c) &\leq C^{\mathcal{A}}(b'_1, \ldots, b'_m, c') && \text{if } c \leq c' \text{ and } b_j \leq b'_j,\ j \in \{1, \ldots, m\}. \end{aligned}$$

For a set S of abstract terms, let $\sqcup S$ denote the least upper bound of S.

We now define the abstraction function for terms. This definition needs an abstraction of *truth values* as an auxiliary construction.

Definition 5.3 (abstraction function α for terms). Let $\tau = C(\bar{u})$ and $\mathcal{N}(\tau) = \langle \sigma_1, \ldots, \sigma_m \rangle$. For the truth values define $\alpha(true) = \texttt{Ter}$ and $\alpha(false) = \texttt{Open}$. If S is a set of terms, define

$$\alpha(S) = \sqcup\{\alpha(t) \mid t \in S\},$$

where $\alpha(t)$ is defined as:

$$\begin{array}{ll} \texttt{Any} & \text{if } t \text{ is a variable,} \\ C^{\mathcal{A}}(\alpha(\mathcal{E}^{\sigma_1}(t, \tau)), \ldots, \alpha(\mathcal{E}^{\sigma_m}(t, \tau)), \alpha(\mathcal{Z}(t, \tau))) & \text{if } t = f_{\langle \tau_1 \ldots \tau_n, \tau \rangle}(t_1, \ldots, t_n). \end{array}$$

Note that this definition is based on the fact that $\alpha(\emptyset) = \texttt{Bot}$. From this it follows that the abstraction of a constant $t = f_{\langle \tau \rangle}$ is $C^{\mathcal{A}}(\texttt{Bot}, \ldots, \texttt{Bot}, \texttt{Ter})$.

The least upper bound of a *set* of abstract terms gives a safe approximation for the instantiation of *all* corresponding concrete terms. *Safe* means that each concrete term is at least as instantiated as indicated by the least upper bound.

Example 5.2. We illustrate Def. 5.3.

$$\begin{array}{ll} \alpha(7) = \texttt{Int}^{\mathcal{A}}(\texttt{Ter}) & (\tau = \texttt{Int}, m = 0, n = 0) \\ \alpha(\texttt{Nil}) & (\tau = \texttt{List(u)}, \mathcal{N}(\tau) = \langle \texttt{u} \rangle, n = 0) \\ \quad = \texttt{List}^{\mathcal{A}}(\alpha(\emptyset), \alpha(\mathcal{Z}(\texttt{Nil}, \tau))) & \\ \quad = \texttt{List}^{\mathcal{A}}(\texttt{Bot}, \texttt{Ter}) & \\ \alpha(\texttt{Cons(7,Nil)}) & (\tau = \texttt{List(u)}, \mathcal{N}(\tau) = \langle \texttt{u} \rangle, n = 2) \\ \quad = \texttt{List}^{\mathcal{A}}(\sqcup\{\alpha(7)\}, \alpha(\mathcal{Z}(\texttt{Cons(7,Nil)}, \tau))) & \\ \quad = \texttt{List}^{\mathcal{A}}(\texttt{Int}^{\mathcal{A}}(\texttt{Ter}), \texttt{Ter}). & \end{array}$$

The table below gives some further examples.

term	**type**	**abstraction**
`x`	`u`	`Any`
`[7,x]`	`List(Int)`	$\texttt{List}^{\mathcal{A}}(\texttt{Any}, \texttt{Ter})$
`[7|x]`	`List(Int)`	$\texttt{List}^{\mathcal{A}}(\texttt{Any}, \texttt{Open})$
`E(7)`	`Nest(Int)`	$\texttt{Nest}^{\mathcal{A}}(\texttt{Int}^{\mathcal{A}}(\texttt{Ter}), \texttt{Ter})$
`[E(7)]`	`List(Nest(Int))`	$\texttt{List}^{\mathcal{A}}(\texttt{Nest}^{\mathcal{A}}(\texttt{Int}^{\mathcal{A}}(\texttt{Ter}), \texttt{Ter}), \texttt{Ter})$
`N([E(7)])`	`Nest(Int)`	$\texttt{Nest}^{\mathcal{A}}(\texttt{Int}^{\mathcal{A}}(\texttt{Ter}), \texttt{Ter})$
`N([E(7),x])`	`Nest(Int)`	$\texttt{Nest}^{\mathcal{A}}(\texttt{Any}, \texttt{Open})$

Note that there is no term of type `Int` whose abstraction is $\texttt{Int}^{\mathcal{A}}(\texttt{Open})$.

The following theorem show that the abstraction captures groundness.

Theorem 5.1 (Proof see [17]). Let S be a set of terms having the same type. Then a variable occurs in an element of S (that is S is non-ground) if and only if `Any` or `Open` occurs in $\alpha(S)$.

5.2 Traversing Abstract Terms

In order to define abstract unification and, in particular, the abstraction of an equation in a program, we require an abstract termination function and abstract extractors similar to those already defined for concrete terms. The type superscript annotation for concrete terms is also useful for abstract terms.

Definition 5.4 (abstract termination function and extractor for σ). Let ϕ and $\tau = C(\bar{u})$ be simple types such that $\tau\Psi \bowtie \phi$ for some Ψ, and $\mathcal{N}(\tau) = \langle \sigma_1, \ldots, \sigma_m \rangle$. Let b be an abstract term for an instance of $\tau\Psi$.
1. Abstract termination function.

$$\begin{array}{ll}
\mathcal{AZ}(b^{\tau\Psi}, \phi) = \texttt{Open} & \text{if } b = \texttt{Any} \\
\mathcal{AZ}(b^{\tau\Psi}, \phi) = \texttt{Ter} & \text{if } b = \texttt{Bot} \\
\mathcal{AZ}(b^{\tau\Psi}, \phi) = c \wedge \bigwedge\limits_{\sigma_j\Psi \bowtie \phi} \mathcal{AZ}(b_j^{\sigma_j\Psi}, \phi) & \text{if } b = C^{\mathcal{A}}(b_1, \ldots, b_m, c).
\end{array}$$

2. Abstract extractor for σ. Let $\sigma \in \mathcal{N}(\phi)$.

$$\begin{array}{ll}
\mathcal{AE}^{\sigma}(b^{\tau\Psi}, \phi) = \texttt{Any} & \text{if } b = \texttt{Any} \\
\mathcal{AE}^{\sigma}(b^{\tau\Psi}, \phi) = \texttt{Bot} & \text{if } b = \texttt{Bot} \\
\mathcal{AE}^{\sigma}(b^{\tau\Psi}, \phi) = \sqcup(\{b_j \mid \sigma_j\Psi = \sigma\} \cup & \\
\qquad\qquad \{\mathcal{AE}^{\sigma}(b_j^{\sigma_j\Psi}, \phi) \mid \sigma_j\Psi \bowtie \phi\}) & \text{if } b = C^{\mathcal{A}}(b_1, \ldots, b_m, c).
\end{array}$$

We omit the superscript $\tau\Psi$ in the expressions $\mathcal{AZ}(b^{\tau\Psi}, \phi)$ and $\mathcal{AE}^{\sigma}(b^{\tau\Psi}, \phi)$ whenever $\phi = \tau$ and Ψ is the identity. In this (very common) case, the abstract termination function is merely a *projection* onto the termination flag of an abstract term (or `Open` if the abstract term is `Any`). Similarly, the abstract extractor for σ is merely a projection onto the j^{th} argument of an abstract term, where $\sigma = \sigma_j$. Note the similarity between the above definition and Thm. 4.2.

Example 5.3.

$$\mathcal{AZ}(\texttt{List}^{\mathcal{A}}(\texttt{Any},\texttt{Ter})^{\texttt{List(Nest(v))}},\ \texttt{Nest(v)}) \ = \ \texttt{Ter} \wedge \mathcal{AZ}(\texttt{Any},\texttt{Nest(v)}) = \texttt{Open}.$$

$$\mathcal{AE}^{\texttt{v}}(\texttt{List}^{\mathcal{A}}(\texttt{Any},\texttt{Ter})^{\texttt{List(Nest(v))}},\ \texttt{Nest(v)}) \ = \ \texttt{Any}.$$

$$\begin{array}{l}
\mathcal{AZ}(\texttt{List}^{\mathcal{A}}(\texttt{Nest}^{\mathcal{A}}(\texttt{Int}^{\mathcal{A}}(\texttt{Ter}),\texttt{Ter}),\texttt{Ter})^{\texttt{List(Nest(v))}},\ \texttt{Nest(v)}) \ = \\
\ \texttt{Ter} \wedge \mathcal{AZ}(\texttt{Nest}^{\mathcal{A}}(\texttt{Int}^{\mathcal{A}}(\texttt{Ter}),\texttt{Ter}),\ \texttt{Nest(v)}) \ = \ \texttt{Ter}.
\end{array}$$

$$\begin{array}{l}
\mathcal{AE}^{\texttt{v}}(\texttt{List}^{\mathcal{A}}(\texttt{Nest}^{\mathcal{A}}(\texttt{Int}^{\mathcal{A}}(\texttt{Ter}),\texttt{Ter}),\texttt{Ter})^{\texttt{List(Nest(v))}},\ \texttt{Nest(v)}) \ = \\
\qquad \mathcal{AE}^{\texttt{v}}(\texttt{Nest}^{\mathcal{A}}(\texttt{Int}^{\mathcal{A}}(\texttt{Ter}),\texttt{Ter}),\ \texttt{Nest(v)}) \ = \ \texttt{Int}^{\mathcal{A}}(\texttt{Ter}).
\end{array}$$

The following theorem states the fundamental relationship between concrete and abstract termination functions and extractors.

Theorem 5.2. Let ϕ and $\tau = C(\bar{u})$ be simple types such that $\tau\Psi \bowtie \phi$ for some Ψ, and $\sigma \in \mathcal{N}(\phi)$. Let $t^{\tau\Psi}$ be a term. Then

$$\alpha(\mathcal{Z}(t^{\tau\Psi}, \phi)) = \mathcal{AZ}(\alpha(t)^{\tau\Psi}, \phi) \tag{1}$$
$$\alpha(\mathcal{E}^{\sigma}(t^{\tau\Psi}, \phi)) = \mathcal{AE}^{\sigma}(\alpha(t)^{\tau\Psi}, \phi) \tag{2}$$

Proof. We only show (2), as the proof for (1) is similar. The proof is by induction on the structure of t. First assume t is a variable x or a constant d. Here we omit the type superscripts because they are irrelevant.

$$\alpha(\mathcal{E}^{\sigma}(x,\phi)) {=} \sqcup \{\alpha(x)\} {=} \mathtt{Any} {=} \mathcal{AE}^{\sigma}(\mathtt{Any},\phi) {=} \mathcal{AE}^{\sigma}(\alpha(x),\phi).$$

$$\alpha(\mathcal{E}^{\sigma}(d,\phi)) {=} \sqcup\, \emptyset {=} \mathtt{Bot} {=} \mathcal{AE}^{\sigma}(C^{\mathcal{A}}(\mathtt{Bot},\dots,\mathtt{Bot},\mathtt{Ter}),\phi) {=} \mathcal{AE}^{\sigma}(\alpha(d),\phi).$$

Now assume t is a compound term. Let $\mathcal{N}(\tau) = \langle \sigma_1, \dots, \sigma_m \rangle$. In the following sequences of equations, $*$ marks steps which use straightforward manipulations such as rearranging least upper bounds or applications of α to sets.

$$\begin{array}{lr}
\mathcal{AE}^{\sigma}(\alpha(t)^{\tau\Psi},\phi) = & \text{(Def. 5.3)} \\
\mathcal{AE}^{\sigma}(C^{\mathcal{A}}(\alpha(\mathcal{E}^{\sigma_1}(t,\tau)),\dots,\alpha(\mathcal{E}^{\sigma_m}(t,\tau)),\alpha(\mathcal{Z}(t,\tau)))^{\tau\Psi},\ \phi) = & \text{(Def. 5.4)} \\
\sqcup(\{\alpha(\mathcal{E}^{\sigma_j}(t,\tau)) \mid \sigma_j\Psi = \sigma\} \cup \{\mathcal{AE}^{\sigma}(\alpha(\mathcal{E}^{\sigma_j}(t,\phi))^{\sigma_j\Psi},\ \phi) \mid \sigma_j\Psi \bowtie \phi\}) = & \\
 & (* \text{ \& hyp.}) \\
\sqcup(\bigcup_{\sigma_j\Psi=\sigma} \{\alpha(\mathcal{E}^{\sigma_j}(t,\tau))\} \qquad \cup \bigcup_{\sigma_j\Psi\bowtie\phi} \{\alpha(\mathcal{E}^{\sigma}(\mathcal{E}^{\sigma_j}(t,\tau)^{\sigma_j\Psi},\ \phi))\}) = & \\
 & (* \text{ \& Thm. 4.3}) \\
\alpha(\mathcal{E}^{\sigma}(t^{\tau\Psi},\phi)). &
\end{array}$$

□

Example 5.4. This illustrates Thm. 5.2 for $\phi = \tau\Psi = \mathtt{List(u)}$ and $\sigma = \mathtt{u}$.

$$\begin{array}{rcl}
\alpha(\mathcal{Z}([7],\mathtt{List(u)})) = & \mathtt{Ter} & = \mathcal{AZ}(\mathtt{List}^{\mathcal{A}}(\mathtt{Int}^{\mathcal{A}}(\mathtt{Ter}),\mathtt{Ter}),\mathtt{List(u)}) \\
\alpha(\mathcal{E}^{\mathtt{u}}([7],\mathtt{List(u)})) = & \mathtt{Int}^{\mathcal{A}}(\mathtt{Ter}) & = \mathcal{AE}^{\mathtt{u}}(\mathtt{List}^{\mathcal{A}}(\mathtt{Int}^{\mathcal{A}}(\mathtt{Ter}),\mathtt{Ter}),\mathtt{List(u)}).
\end{array}$$

5.3 Abstract Compilation

We now show how the abstract domains can be used in the context of *abstract compilation*. We define an abstract program and show that it is a safe approximation of the concrete program with respect to the usual operational semantics.

In a (normal form) program, each unification is made explicit by an equation. We now define an abstraction of such an equation. For an equation of the form $x = f(y_1, \dots, y_n)$, the abstraction is an atom of the form $f_{\mathrm{dep}}(b, b_1, \dots, b_n)$, where f_{dep} is a predicate defined in the abstract program.

Definition 5.5 (f_{dep}). Let $f_{\langle\tau_1\ldots\tau_n,\tau\rangle} \in \Sigma_f$ where $\tau = C(\bar{u})$ and $\mathcal{N}(\tau) = \langle\sigma_1,\ldots,\sigma_m\rangle$. Then $f_{\text{dep}}(b,b_1,\ldots,b_n)$ **holds** if

$$
\begin{aligned}
b &= C^{\mathcal{A}}(a_1,\ldots,a_m,c) \quad \text{where} \\
a_j &= \sqcup\left(\{b_i \mid \tau_i = \sigma_j\} \cup \{\mathcal{A}\mathcal{E}^{\sigma_j}(b_i^{\tau_i},\tau) \mid {\scriptstyle \tau_i \bowtie \tau}\}\right) \qquad \text{for all } {\scriptstyle j\in\{1,\ldots,m\}} \qquad (1) \\
c &= \bigwedge_{\tau_i \bowtie \tau} \mathcal{A}\mathcal{Z}(b_i^{\tau_i},\tau) \qquad (2)
\end{aligned}
$$

Example 5.5. To give an idea of how Def. 5.5 translates into code, consider `Cons`. Assuming that `Lub`(a,b,c) holds if $c = \sqcup\{a,b\}$, one clause for $\texttt{Cons}_{\text{dep}}$ might be:

```
Cons_dep(List_a(c,Ter),b,List_a(a,Ter)) <-
  Lub(a,b,c).
```

The following theorem shows that f_{dep} correctly captures the dependency between $\alpha(f(t_1,\ldots,t_n))$ and $\alpha(t_1),\ldots,\alpha(t_n)$.

Theorem 5.3. If $t = f(t_1,\ldots,t_n)$ then $f_{\text{dep}}(\alpha(t),\alpha(t_1),\ldots,\alpha(t_n))$ holds.

Proof. Suppose $\mathcal{N}(\tau) = \langle\sigma_1,\ldots,\sigma_m\rangle$ and $\tau = C(\bar{u})$. By Def. 5.3

$$\alpha(t) = C^{\mathcal{A}}(\alpha(\mathcal{E}^{\sigma_1}(t,\tau)),\ldots,\alpha(\mathcal{E}^{\sigma_m}(t,\tau)),\alpha(\mathcal{Z}(t,\tau))).$$

We show that (1) in Def. 5.5 holds. For each $\sigma_j \in \mathcal{N}(\tau)$,

$$
\begin{aligned}
&\alpha(\mathcal{E}^{\sigma_j}(t,\tau)) \\
&\quad = \alpha(\{t_i \mid \tau_i = \sigma_j\} \quad \cup \bigcup_{\tau_i \bowtie \tau} \mathcal{E}^{\sigma_j}(t_i^{\tau_i},\tau)) && \text{(Thm. 4.2)} \\
&\quad = \sqcup\left(\{\alpha(t_i) \mid \tau_i = \sigma_j\} \cup \{\alpha(\mathcal{E}^{\sigma_j}(t_i^{\tau_i},\tau)) \mid {\scriptstyle \tau_i \bowtie \tau}\}\right) && \text{(moving } \alpha \text{ inwards)} \\
&\quad = \sqcup\left(\{\alpha(t_i) \mid \tau_i = \sigma_j\} \cup \{\mathcal{A}\mathcal{E}^{\sigma_j}(\alpha(t_i)^{\tau_i},\tau) \mid {\scriptstyle \tau_i \bowtie \tau}\}\right) && \text{(Thm. 5.2).}
\end{aligned}
$$

Equation (2) in Def. 5.5 is proven in a similar way. □

Definition 5.6 (abstraction $\aleph$ of a program). For a normal form equation e define

$$\aleph(e) = \begin{cases} e & \text{if } e \text{ is of the form } x = y \\ f_{\text{dep}}(x,y_1,\ldots,y_n) & \text{if } e \text{ is of the form } x = f(y_1,\ldots,y_n). \end{cases}$$

For a normal form atom a and clause $K = h \leftarrow g_1 \wedge \cdots \wedge g_l$ define

$$
\begin{aligned}
\aleph(a) &= a \\
\aleph(K) &= \aleph(h) \leftarrow \aleph(g_1) \wedge \cdots \wedge \aleph(g_l).
\end{aligned}
$$

For a program $P = \langle L, S\rangle$ define

$$\aleph(P) = \{\aleph(K) \mid K \in S\} \cup \{f_{\text{dep}}(a,a_1,\ldots,a_n) \mid f_{\text{dep}}(a,a_1,\ldots,a_n) \text{ holds}\}.$$

Example 5.6. In the following we give the usual recursive clause for `Append` in normal form and its abstraction.

```
%concrete clause                    %abstract clause
Append(xs,ys,zs) <-                 Append(xs,ys,zs) <-
  xs = [x|x1s] &                      Cons_dep(xs,x,x1s) &
  zs = [x|z1s] &                      Cons_dep(zs,x,z1s) &
  Append(x1s,ys,z1s).                 Append(x1s,ys,z1s).
```

We now define the operational semantics of concrete and abstract programs. We assume a fixed language L and program $P = \langle L, S\rangle$, and a left-to-right computation rule. A *program state* is a tuple $\langle G, \Theta\rangle$ where G is a query and Θ a substitution. It is an initial state if Θ is empty. We write $C \in_{\approx} S$ if C is a renamed variant of a clause in S.

Definition 5.7 (reduces to). The relation $\xrightarrow{P}$ ("reduces to") between states is defined by the following rules:

$$\langle h_1 : \cdots : h_l, \Theta\rangle \xrightarrow{P} \langle h_2 : \cdots : h_l, \Theta\Theta'\rangle \quad \text{if } h_1 \text{ is `} x = t\text{' and } x\Theta\Theta' = t\Theta\Theta' \qquad (1)$$

$$\langle h_1 : \cdots : h_l, \Theta\rangle \xrightarrow{P} \langle G : h_2 : \cdots : h_l, \Theta\Theta'\rangle \quad \text{if } h \leftarrow G \in_{\approx} S \text{ and } h\Theta\Theta' = h_1\Theta\Theta' \quad (2)$$

$\xrightarrow{P}{}^{j}$ for $j \geq 0$ and $\xrightarrow{P}{}^{*}$ are defined in the usual way. If for an initial query G,

$$\langle G, \emptyset\rangle \xrightarrow{P}{}^{*} \langle p(x_1, \ldots, x_n) : H, \Theta\rangle \xrightarrow{P}{}^{*} \langle H, \Theta'\rangle,$$

we call $p(x_1, \ldots, x_n)\Theta$ a *call pattern* and $p(x_1, \ldots, x_n)\Theta'$ an *answer pattern* for p.

Note that this notion of "reduces" with arbitrary unifier is considered in [13].

The next theorem shows that for all call and answer patterns, which may arise in a derivation of a concrete program, there are corresponding patterns in a derivation of the abstract program.

Theorem 5.4. Let H, H' be queries, Θ a substitution and $j \geq 0$. If $\langle H, \emptyset\rangle \xrightarrow{P}{}^{j} \langle H', \Theta\rangle$, then $\langle \aleph(H), \emptyset\rangle \xrightarrow{\aleph(P)}{}^{j} \langle \aleph(H'), \Theta^{\alpha}\rangle$, where $\Theta^{\alpha} = \{x/\alpha(x\Theta) \mid x \in \mathit{dom}(\Theta)\}$.

Proof. By Def. 5.7, $\langle H, \emptyset\rangle \xrightarrow{P}{}^{j} \langle H', \Theta\rangle$ if and only if $\langle H, \Theta\rangle \xrightarrow{P}{}^{j} \langle H', \Theta\rangle$, and likewise for $\aleph(P)$. Therefore it is enough to show that for all $j \geq 0$

$$\langle H, \Theta\rangle \xrightarrow{P}{}^{j} \langle H', \Theta\rangle \qquad \text{implies} \qquad \langle \aleph(H), \Theta^{\alpha}\rangle \xrightarrow{\aleph(P)}{}^{j} \langle \aleph(H'), \Theta^{\alpha}\rangle. \qquad (3)$$

The proof is by induction on j. The base case $j = 0$ holds since $\langle \aleph(H), \Theta^{\alpha}\rangle \xrightarrow{\aleph(P)}{}^{0} \langle \aleph(H), \Theta^{\alpha}\rangle$. For the induction step, assume (3) holds for some $j \geq 0$. We show that for every query H''

$$\langle H, \Theta\rangle \xrightarrow{P}{}^{j+1} \langle H'', \Theta\rangle \qquad \text{implies} \qquad \langle \aleph(H), \Theta^{\alpha}\rangle \xrightarrow{\aleph(P)}{}^{j+1} \langle \aleph(H''), \Theta^{\alpha}\rangle.$$

If $\langle H, \Theta \rangle \xrightarrow{P}{}^{j+1} \langle H'', \Theta \rangle$ is *false*, the result is trivial. If $\langle H, \Theta \rangle \xrightarrow{P}{}^{j+1} \langle H'', \Theta \rangle$, then

$$\begin{array}{llll} \langle H, \Theta \rangle & \xrightarrow{P}{}^{j} \langle H', \Theta \rangle & \xrightarrow{P} \langle H'', \Theta \rangle & \text{for some query } H', \text{ and} \\ \langle \aleph(H), \Theta^\alpha \rangle & \xrightarrow{\aleph(P)}{}^{j} \langle \aleph(H'), \Theta^\alpha \rangle & & \text{by hypothesis.} \end{array}$$

It only remains to be shown that $\langle \aleph(H'), \Theta^\alpha \rangle \xrightarrow{\aleph(P)} \langle \aleph(H''), \Theta^\alpha \rangle$. We distinguish whether Rule (1) or (2) of Def. 5.7 was used for the step $\langle H', \Theta \rangle \xrightarrow{P} \langle H'', \Theta \rangle$.

(1): $H' = h_1 : \cdots : h_l$ where h_1 is '$x = t$', and $t = y$ or $t = f(x_1, \ldots, x_n)$. In the first case $\aleph(h_1) = h_1$. Since $x\Theta = y\Theta$, it follows that $\{x/\alpha(x\Theta), y/\alpha(x\Theta)\} \subseteq \Theta^\alpha$ and therefore $x\Theta^\alpha = y\Theta^\alpha$. Thus $\langle \aleph(H'), \Theta^\alpha \rangle \xrightarrow{\aleph(P)} \langle \aleph(H''), \Theta^\alpha \rangle$ by Rule (1). In the second case $\aleph(h_1) = f_{\text{dep}}(x, x_1, \ldots, x_n)$. Since $x\Theta = f(x_1\Theta, \ldots, x_n\Theta)$,

$$\{x/\alpha(f(x_1\Theta, \ldots, x_n\Theta)), x_1/\alpha(x_1\Theta), \ldots, x_n/\alpha(x_n\Theta)\} \subseteq \Theta^\alpha.$$

Thus, by Thm. 5.3, $f_{\text{dep}}(x, x_1, \ldots, x_n)\,\Theta^\alpha$ holds so that $f_{\text{dep}}(x, x_1, \ldots, x_n)\,\Theta^\alpha \in \aleph(P)$ by Def. 5.6. Thus $\langle \aleph(H'), \Theta^\alpha \rangle \xrightarrow{\aleph(P)} \langle \aleph(H''), \Theta^\alpha \rangle$ by Rule (2).

(2): $H' = h_1 : \cdots : h_l$ where $h \leftarrow G \in_\approx S$ and $h\Theta = h_1\Theta$. By Def. 5.6, $\aleph(h_1 \leftarrow G) \in_\approx \aleph(P)$. Furthermore $\aleph(h)$ has the form $p(\bar{x})$, and $\aleph(h_1)$ has the form $p(\bar{y})$. Since $\bar{x}\Theta = \bar{y}\Theta$ it follows that $p(\bar{x})\,\Theta^\alpha = p(\bar{y})\,\Theta^\alpha$. □

6 Implementation and Results

From now on we refer to the abstract domains defined in this paper as *typed domains*. We have implemented our mode analysis for object programs in Gödel. This implementation naturally falls into two stages: In the first stage, the language declarations are analysed in order to construct the typed domains, and the program clauses are abstracted. In the second stage, the abstract program is evaluated using standard abstract compilation techniques.

We have implemented the first stage in Gödel, using the Gödel meta-programming facilities. Gödel meta-programming is slow, but this first stage scales well, as the time for abstracting the clauses of a program is linear in their number. Analysing the type declarations is not a problem in practice. We have analysed contrived, complex type declarations within a couple of seconds.

The second stage was implemented in Prolog, so that an existing analyser could be used. Abstract programs produced by the first stage were transformed into Prolog. All call and answer patterns, which may arise in a derivation of an abstract program for a given query, are computed by the analyser. By Thm. 5.4, these patterns correspond to patterns in the derivation of the concrete program. For example a call $\mathtt{p(Any, Int^{\mathcal{A}}(Ter))}$ in the abstract program indicates that there may be a call `p(x,7)` in the concrete program.

We now demonstrate the precision of the typed domain for `Table(Int)`. The arguments of the predicate `Insert` represent: a table t, a key k, a value v, and a table obtained from t by inserting the node whose key is k and whose value

Table 1. Some call and answer patterns for `Insert`

$\texttt{Insert}(\texttt{Tab}^{\mathcal{A}}(\texttt{Int}^{\mathcal{A}}, \texttt{Bal}^{\mathcal{A}}, \texttt{Str}^{\mathcal{A}}, \texttt{Ter}), \texttt{Str}^{\mathcal{A}}, \texttt{Int}^{\mathcal{A}}, \texttt{Any})$ *leads to answer pattern* $\texttt{Insert}(\texttt{Tab}^{\mathcal{A}}(\texttt{Int}^{\mathcal{A}}, \texttt{Bal}^{\mathcal{A}}, \texttt{Str}^{\mathcal{A}}, \texttt{Ter}), \texttt{Str}^{\mathcal{A}}, \texttt{Int}^{\mathcal{A}}, \texttt{Tab}^{\mathcal{A}}(\texttt{Int}^{\mathcal{A}}, \texttt{Bal}^{\mathcal{A}}, \texttt{Str}^{\mathcal{A}}, \texttt{Ter}))$.
$\texttt{Insert}(\texttt{Tab}^{\mathcal{A}}(\texttt{Int}^{\mathcal{A}}, \texttt{Bal}^{\mathcal{A}}, \texttt{Str}^{\mathcal{A}}, \texttt{Ter}), \texttt{Str}^{\mathcal{A}}, \texttt{Any}, \texttt{Any})$ *leads to answer pattern* $\texttt{Insert}(\texttt{Tab}^{\mathcal{A}}(\texttt{Int}^{\mathcal{A}}, \texttt{Bal}^{\mathcal{A}}, \texttt{Str}^{\mathcal{A}}, \texttt{Ter}), \texttt{Str}^{\mathcal{A}}, \texttt{Any}, \texttt{Tab}^{\mathcal{A}}(\texttt{Any}, \texttt{Bal}^{\mathcal{A}}, \texttt{Str}^{\mathcal{A}}, \texttt{Ter}))$.

is v. Table 1 shows some initial call patterns and the answer pattern that is inferred for each call pattern. For readability, we have used some abbreviations and omitted the termination flag for types `Integer`, `Balance` and `String`.

Clearly, inserting a ground node into a ground table gives a ground table. This can be inferred with the typed domains, but it could also be inferred using a domain which can only distinguish between ground and non-ground terms [4]. Now consider the insertion of a node with an *uninstantiated* value into a ground table. With typed domains, it can be inferred that the result is still a table but whose values may be uninstantiated.

We used a modified form of the analyser of [8] running on a Sun SPARC Ultra 170. The analysis times for the two example analyses using `Insert` were were 0.81 seconds and 2.03 seconds, respectively. Comparing this to an analysis using a domain which can only distinguish ground and non-ground terms, the times were 0.09 seconds and 1.57 seconds, respectively. Apart from `Tables`, we also analysed some small programs, namely `Append`, `Reverse`, `Flatten` (from the `Nests` module), `TreeToList`, `Qsort`, and `Nqueens`. For these, all analysis times were below 0.03 seconds and thus too small to be very meaningful.

Our experience is that the domain operations, namely to compute the least upper bound of two abstract terms, are indeed the bottleneck of the analysis. Therefore it is crucial to avoid performing these computations unnecessarily. Also one might compromise some of the precision of the analysis by considering widenings [6] for the sake of efficiency. In order to conduct more experiments, one would need a suite of bigger typed logic programs. A formal comparison between analyses for typed logic programs and untyped ones is of course difficult.

7 Discussion and Related Work

We have presented a general domain construction for mode analysis of typed logic programs. This analysis gives more accurate information than one based on a ground/non-ground domain [4]. For common examples (lists, binary trees), our formalism is simple and yields abstract domains that are comparable to the domains in [3]. The novelty is that the construction is described for arbitrary types. In contrast, in [3], an abstract domain for obtaining this degree of precision for, say, the types in the `Tables` module, would have to be hand-crafted.

The fundamental concepts of this work are *recursive type* and *non-recursive subterm type*, which are generalisations of ideas presented in [3] for lists. The resulting abstract domains are entirely in the spirit of [3, 5] and we believe that they provide the highest degree of precision that a generic domain construction should provide. Even if type declarations that require the full generality of our formalism are rare, we think that our work is an important contribution because it helps to understand other, more ad-hoc and pragmatic domain constructions as instances of a general theory. One could always simplify or prune down our abstract domains for the sake of efficiency.

In its full generality the formalism is, admittedly, rather complex. This is mainly due to function declarations where the range type occurs again as a *proper* sub"term" of an argument type, such as the declaration of `N` in Ex. 2.2. This phenomenon occurs in the declarations for *rose trees* [14], that is, trees where the number of children of each node is not fixed. One should note that while the theory which allows for a domain construction for, say, `Nest(Int)` is conceptually complex, the computational complexity of the domain operations for `Nest(Int)` is lower than for, say, `List(List(List(Int)))`. In short, the complexity of the abstract domains depends on the complexity of the type declarations.

We have built on the ideas presented in [5] for untyped languages. Notably the title of [5] says that *type*, not *mode*, dependencies are derived. Even in an untyped language such as Prolog, one can define types as sets of terms given by some kind of "declaration", just as in a typed language [1]. In this case type analysis (that is, inferring that an argument is instantiated to a term *of a certain type*) is inseparable from mode analysis. It seems that [5] provides a straightforward domain construction for *arbitrary* types, but this is not the case. It is not specified what kind of "declarations" are implied, but the examples and theory suggest that all types are essentially lists and trees. The `Tables` and `Nests` examples given in Sect. 2 are not captured.

Recursive modes [21] characterise that the left spine, right spine, or both, of a term are instantiated. The authors admit that this may be considered an ad-hoc choice, but on the other hand, they present good experimental results. They do not assume a typed language and thus cannot exploit type declarations in order to provide a more generic concept of *recursive modes*, as we have done by the concept of *termination*.

A complex system for type analysis of Prolog is presented in [23]. As far as we can see, this system is not in a formal sense stronger or weaker that our mode analysis. The domain `Pat(Type)` used there is infinite, so that widenings have to be introduced to ensure finiteness, and "the design of widening operators is experimental in nature" [23]. In contrast, we exploit the type declarations to construct domains that are inherently finite and whose size is immediately dictated by the complexity of the type declarations.

Mercury [19] has a strong mode system based on *instantiation states*. These are assertions of how instantiated a term is. An instantiation state is similar to an abstract term. Indeed, given some type declarations, it is possible to define an instantiation state in Mercury syntax which, while not being exactly the same, is comparable in precision to an abstract term in our formalism. The difference is that for a given type, there are potentially infinitely many instantiation states.

The current Mercury implementation does not support instantiation states in their full generality, although a version supporting partially instantiated data-structures is being developed. Within the limits of the expressiveness of the mode system, Mercury does a combination of mode analysis and mode checking of modes declared by the user.

Even if instantiation states were supported in their full generality, the potentially infinite number of instantiation states means that mode inference must always be approximate. Since our abstract terms formalise what might be called a "reasonable" degree of precision, we believe that our proposal could serve as a basis for this approximation. One could envisage a Mercury implementation doing a combination of mode inference and checking, based on the set of modes which is expressible using our abstract domains. Hence our domains could also be used to *declare* modes.

The mode system in Mercury is based on [18], where the Simple Range Condition and the Reflexive Condition that we impose are not explicitly required. However, [18] does not define the type system precisely, instead referring to [15], whose formal results have been shown to be incorrect [16]. It is therefore difficult to assess whether that approach would work for programs which violate these conditions. We know of no real Gödel programs that violate either of the Simple Range or Reflexive Conditions. We have found that violating the Reflexive Condition raises fundamental questions about decidability in typed languages, which seem to be related to the concept of *polymorphic recursion* [11, 12].

There is another potential application of our work. In Gödel, the delay declarations which state that a predicate is delayed until an argument (or a subterm of the argument) is ground or non-variable, cannot describe the behaviour of the Gödel system predicates precisely. We have observed that, typically, the degree of instantiation for a Gödel system predicate to run safely without delaying could be specified by an abstract term in our typed domains. Thus they could provide a good basis for declaring conditions for delaying.

Our approach may also be applicable to untyped languages, if we have information at hand that is similar to type declarations. Such information might be obtained by inferring declarations [2] or from declarations as comments [20]. Certainly our analysis would then regain aspects of *type* rather than *mode* inference, which it had lost by transferring the approach to typed languages.

Acknowledgements

We thank Tony Bowers, Henning Christiansen, Bart Demoen, Andrew Heaton, Fergus Henderson, Jonathan Martin and Lambert Meertens for helpful discussions and comments. Jan–Georg Smaus was supported by EPSRC Grant No. GR/K79635.

References

1. A. Aiken and T. K. Lakshman. Directional type checking of logic programs. In *SAS '94*, pages 43–60. Springer-Verlag, 1994.

2. H. Christiansen. Deriving declarations from programs. Technical report, Roskilde University, P.O.Box 260, DK-4000 Roskilde, 1997.
3. M. Codish and B. Demoen. Deriving polymorphic type dependencies for logic programs using multiple incarnations of Prop. In *SAS'94*, pages 281–297. Springer-Verlag, 1994.
4. M. Codish and B. Demoen. Analyzing logic programs using "PROP"-ositional logic programs and a Magic Wand. *Journal of Logic Programming*, 25(3):249–274, 1995.
5. M. Codish and V. Lagoon. Type dependencies for logic programs using ACI-unification. In *Israeli Symposium on Theory of Computing and Systems*, pages 136–145. IEEE Press, 1996.
6. P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *PLILP'92*, LNCS, pages 269–295. Springer-Verlag, 1992.
7. J. Gallagher, D. Boulanger, and H. Sağlam. Practical model-based static analysis for definite logic programs. In J. W. Lloyd, editor, *ILPS'95*, pages 351–365. MIT Press, 1995.
8. A.J. Heaton, P.M. Hill, and A.M. King. Analysing logic programs with delay for downward-closed properties. In N.E. Fuchs, editor, *LOPSTR'97*, LNCS. Springer-Verlag, 1997.
9. P.M. Hill and A. King. Determinacy and determinacy analysis. *Journal of Programming Languages*, 5(1):135–171, 1997.
10. P.M. Hill and J.W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.
11. S. Kahrs. Limits of ML-definability. In H. Kuchen and S. D. Swierstra, editors, *PLILP'96*, LNCS, pages 17–31. Springer-Verlag, 1996.
12. A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type recursion in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, 1993.
13. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
14. L. Meertens. First steps towards the theory of rose trees. CWI, Amsterdam; IFIP Working Group 2.1 working paper 592 ROM-25, 1988.
15. A. Mycroft and R. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.
16. F. Pfenning, editor. *Types in Logic Programming*, chapter 1. MIT Press, 1992.
17. J.-G. Smaus, P. M. Hill, and A. M. King. Mode analysis domains for typed logic programs. Technical Report 2000.06, School of Computer Studies, University of Leeds, 2000. © Springer-Verlag.
18. Z. Somogyi. A system of precise modes for logic programs. In *ICLP'87*, pages 769–787. MIT Press, 1987.
19. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, November 1996.
20. K. Stroetmann and T. Glaß. A semantics for types in Prolog: The type system of PAN version 2.0. Technical report, Siemens AG, München, Germany, 1995.
21. Jichang Tan and I-Peng Lin. Recursive modes for precise analysis of logic programs. In *ILPS'97*, pages 277–290. MIT Press, 1997.
22. M. van Emden. AVL tree insertion: A benchmark program biased towards Prolog. *Logic Programming Newsletter 2*, 1981.
23. P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of Prolog using type graphs. Technical Report CS-93-52, Brown University Box 1910, Providence, RI 02912, November 1993.

$$
\begin{tikzcd}
\cdots & & C_3(\bar{u}_3) \arrow[d, "\Psi_3"'] & & C_2(\bar{u}_2) \arrow[d, "\Psi_2"'] & & C_1(\bar{u}_1) \arrow[d, "\Psi_1"'] \\
\cdots & \mathrm{NRS} \arrow[ur] & = & \mathrm{NRS} \arrow[ur] & = & \mathrm{NRS} \arrow[ur] & = \\
\tau_4 \arrow[ur, dash] & & \tau_3 \arrow[u, "\Psi_2"] \arrow[ur, dash] & & \tau_2 \arrow[u, "\Psi_1"] \arrow[ur, dash] & & \tau_1 \arrow[u, "\Psi_0"]
\end{tikzcd}
$$

Fig. 3. The sequence of non-recursive subterm types

A Proofs

The following lemma states that the relation $\lhd$ is closed under instantiation of its arguments.

Lemma A.1. Let σ, ϕ be types and Ψ a type substitution. If $\sigma \lhd \phi$ then $\sigma\Psi \lhd \phi\Psi$. If $\sigma \lhd^* \phi$ then $\sigma\Psi \lhd^* \phi\Psi$.

Proof. For the first statement, there is $f_{\langle\tau_1\ldots\tau_n,\tau\rangle} \in \Sigma_f$ and a type substitution Ψ' such that for some $i \in \{1,\ldots,n\}$, $\tau_i\Psi' = \sigma$ and $\tau\Psi' = \phi$. Consequently $\tau_i\Psi'\Psi = \sigma\Psi$ and $\tau\Psi'\Psi = \phi\Psi$, so $\sigma\Psi \lhd \phi\Psi$. The second statement follows from the first. $\square$

The next lemma ensures that the abstract domains are well-defined. It states that any sequence of non-recursive subterm types terminates.

Lemma A.2. Let $\tau \in T(\Sigma_\tau, U) \setminus U$ and $\Gamma \subseteq \Sigma_\tau$. Let I be a non-empty index set (finite or infinite) starting at 1 and $\{(C_i(\bar{u}_i), \tau_i, \Psi_i) \mid i \in I\}$ a sequence where $C_1 \in \Gamma$, $\tau_1 = C_1(\bar{u}_1)\Psi_1 = \tau$, $dom(\Psi_1) \subseteq \bar{u}_1$ and, for each $i \in I$ where $i > 1$:

- $C_i \in \Gamma$, $dom(\Psi_i) \subseteq \bar{u}_i$ and $C_i(\bar{u}_i)\Psi_i = \tau_i\Psi_{i-1}$,
- $\tau_i \in T(\Gamma, U)$ and $\tau_i \in \mathcal{N}(C_{i-1}(\bar{u}_{i-1}))$.

Then I and hence $\{(C_i(\bar{u}_i), \tau_i, \Psi_i) \mid i \in I\}$ is finite.

Proof. Let Ψ_0 be the identity substitution. The sequence is illustrated in Fig. 3. First note that, by Lemma A.1 and Def. 4.2, for each $i \in I$ where $i \geq 2$, we have $\tau_i\Psi_{i-1} \lhd^* \tau_{i-1}\Psi_{i-2}$. Thus, for all $i, j \in I$ where $i > j$, $\tau_i\Psi_{i-1} \lhd^* \tau_j\Psi_{j-1}$.

Let $d(\rho)$ be the number of occurrences of constructors in a type ρ. If $\Gamma_0 \subseteq \Sigma_\tau$, define

$$D(\Gamma_0, \rho) = d(\rho) + \sum_{C \in \Gamma_0} \left(\sum_{\sigma \in \mathcal{N}(C(\bar{u}))} d(\sigma) \right).$$

The proof is by induction on $D(\Gamma, \tau)$. Since $\tau \notin U$, it follows that $D(\Gamma, \tau) \geq 1$. If $D(\Gamma, \tau) = 1$, then $\tau = C_1(\bar{u}_1)$, $\mathcal{N}(C_1(\bar{u}_1)) \subseteq U$ and $|I| \leq 2$.

Suppose that $D(\Gamma, \tau) = M > 1$. Assume that, for all types ρ and sets of constructors $\Gamma_0 \subseteq \Gamma$ such that $D(\Gamma_0, \rho) < M$, the result holds. Since the result

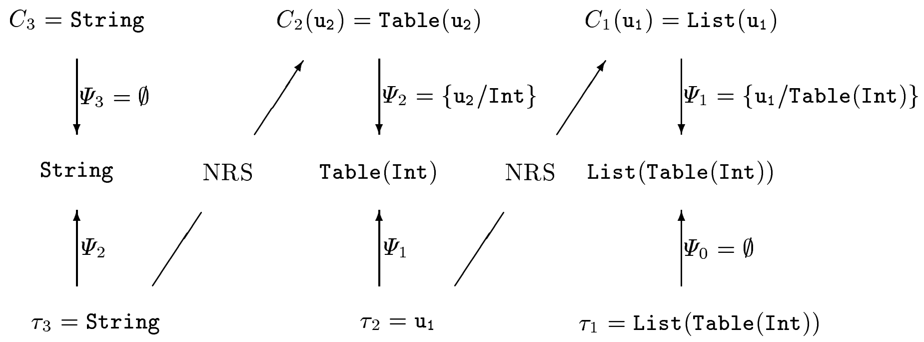

Fig. 4. An example of the sequence of non-recursive subterm types

obviously holds if $|I| \leq 2$, suppose $|I| > 2$ so that τ_2 is not a parameter. Consider the sequence $\{(C_i(\bar{u}_i), \tau_i, \Psi'_i) \mid i \in I'\}$ where I' is an index set starting at 2, Ψ'_1 is the identity substitution and, for each $i \in I'$, we have $C_i(\bar{u}_i)\Psi'_i = \tau_i \Psi'_{i-1}$. Since $\tau_i \in \mathcal{N}(C_{i-1}(\bar{u}_{i-1}))$, $\Psi'_i\Psi_1 = \Psi_i$ for each $i \in I'$. As in the first paragraph, for each $i \in I'$, $\tau_i \Psi'_{i-1} \lhd^* \tau_2$. However, $\tau_2 \in \mathcal{N}(C_1(\bar{u}_1))$. Thus, by the Reflexive Condition and Lemma 4.1, for each $i \in I'$, we have $C_i \neq C_1$. Thus, for each $i \in I'$, we have $C_i \in \Gamma'$ where $\Gamma' = \Gamma \setminus \{C_1\}$. However,

$$D(\Gamma', \tau_2) = d(\tau_2) + D(\Gamma, \tau) - d(\tau) - \sum_{\sigma \in \mathcal{N}(C_1(\bar{u}))} d(\sigma).$$

Hence, as $d(\tau) > 0$ and $\tau_2 \in \mathcal{N}(C_1(\bar{u}))$, $D(\Gamma', \tau_2) < M$ and we can use the induction hypothesis. Hence I' is finite.

Assume now that I' is maximal wrt. to the above conditions and that $|I'| = N'$ and suppose $K = N' + 1 \in I$. (If $K \notin I$, then, as I' is finite, I is finite.) Then $\tau_K \Psi'_{K-1} = u$ where u is parameter since, if $\tau_K \Psi'_{K-1} = C_K(\bar{u}_K)\Psi'_K$, then K also satisfies the above conditions so that I' is not maximal. Thus Ψ'_{K-1} is the identity substitution and $\tau_K = u$. By the transparency condition, since $\tau_K \lhd^* C_1(\bar{u}_1)$, $u \in \bar{u}_1$. As $\Psi_{K-1} = \Psi'_{K-1}\Psi_1$, we have $\Psi_{K-1} = \Psi_1$ and $\tau_K \Psi_{K-1} \in \bar{u}_1 \Psi_1$. Hence $d(\tau_K \Psi_{K-1}) < d(\tau)$ so that

$$D(\Gamma, \tau_K \Psi_{K-1}) < D(\Gamma, \tau).$$

Hence, the inductive hypothesis can be applied to the remaining sequence starting at τ_K. Thus the subsequence starting at τ_K is finite and therefore the complete sequence starting at τ is finite. $\square$

Example A.1. Figure 4 gives an example of a sequence of types as constructed in Lemma A.2. The abstract domain for `List(Table(Int))` is defined in terms of the abstract domain for `Table(Int)`, and the abstract domain for `Table(Int)` is defined in terms of the abstract domain for `String`. Therefore it is crucial that any such chain is finite.

The following lemmas are needed in the proof of Thm. 4.3.

Lemma A.3. Let ϕ be a type, Ψ a type substitution, and t a term having a type which is an instance of $\phi\Psi$. If s^τ is a subterm of t^ϕ, then s has a type which is an instance of $\tau\Psi$.

Proof. Induction on the depth of subterms. □

Lemma A.4. Let $\sigma_1, \sigma_2, \sigma_3$ be types. If $\sigma_1 \bowtie \sigma_2$ and $\sigma_2\Psi \bowtie \sigma_3$ for some type substitution Ψ then $\sigma_1\Psi \bowtie \sigma_3$.

Proof. By Lemma A.1 it follows that $\sigma_1\Psi \lhd^* \sigma_3$ and $\sigma_3 \lhd^* \sigma_1\Psi$. □

Theorem 4.3. Let ϕ and τ be simple types such that $\tau\Psi \bowtie \phi$ for some Ψ, let t be a term having a type which is an instance of $\tau\Psi$, and $\sigma \in \mathcal{N}(\phi)$. Then

$$\mathcal{Z}(t^{\tau\Psi}, \phi) = \qquad \mathcal{Z}(t, \tau) \wedge \bigwedge_{\substack{\rho \in \mathcal{N}(\tau) \\ \rho\Psi \bowtie \phi}} \mathcal{Z}(\mathcal{E}^\rho(t, \tau)^{\rho\Psi}, \phi) \tag{1}$$

$$\mathcal{E}^\sigma(t^{\tau\Psi}, \phi) = \bigcup_{\substack{\rho \in \mathcal{N}(\tau) \\ \rho\Psi = \sigma}} \mathcal{E}^\rho(t, \tau) \cup \bigcup_{\substack{\rho \in \mathcal{N}(\tau) \\ \rho\Psi \bowtie \phi}} \mathcal{E}^\sigma(\mathcal{E}^\rho(t, \tau)^{\rho\Psi}, \phi) \tag{2}$$

Proof. The proof consists of four parts. In Part 1, we define a number of sets of subterms of t. We then show six propositions which say that each expression occurring in (1) and (2) can be expressed in terms of these sets. In Part 2 we show how the left and right hand sides of both (1) and (2) can be related using these sets. This is then used in Part 3 to show (1), and in Part 4 to show (2).

Part 1: To avoid confusion between the many symbols occurring in the proof, keep in mind that ϕ, τ, σ and Ψ occur in the statement and thus are *fixed*. We use f as an abbreviation for $f_{\langle \tau'_1 \dots \tau'_n, \tau' \rangle}$ (*not* $f_{\langle \tau_1 \dots \tau_n, \tau \rangle}$, as earlier in this paper), and $\bar{r}$ to denote $(r_1, \dots, r_n)$. Superscripts are omitted where irrelevant. Define

$$\begin{aligned} R &= \{r^\omega \mid r^\omega \text{ is a } \phi\text{–recursive subterm of } t^{\tau\Psi}\} \\ S &= \{r_i \mid f(\bar{r})^{\tau'\Psi'} \in R \text{ and } \tau'_i\Psi' = \sigma\} \\ A &= \{r^\omega \mid r^\omega \text{ is a } \tau\text{–recursive subterm of } t^\tau\}. \end{aligned}$$

Note that, by Lemma A.3, each $r^\omega \in A$ has a type which is an instance of $\omega\Psi$. Furthermore for all $\rho \in \mathcal{N}(\tau)$ define

$$B^\rho = \{r_i \mid f(\bar{r})^{\tau'\Psi'} \in A \text{ and } \tau'_i\Psi' = \rho\}.$$

Note that, by Lemma A.3, each $r_i \in B^\rho$ has a type which is an instance of $\tau'_i\Psi'\Psi$ $(= \rho\Psi)$. For all $\rho \in \mathcal{N}(\tau)$ with $\rho\Psi \bowtie \phi$ define

$$\begin{aligned} C^\rho &= \{r^\omega \mid r^\omega \text{ is a } \phi\text{-recursive subterm of some } s^{\rho\Psi}, s \in B^\rho\} \\ D^\rho &= \{r_i \mid f(\bar{r})^{\tau'\Psi'} \in C^\rho \text{ and } \tau'_i\Psi' = \sigma\}. \end{aligned}$$

S1-S6 state how these sets relate to the computations of (1) and (2).

S1 $\mathcal{Z}(t^{\tau\Psi}, \phi) = false$ if and only if $vars(R) \neq \emptyset$.
S2 $\mathcal{Z}(t, \tau) = false$ if and only if $vars(A) \neq \emptyset$.
S3 $\mathcal{E}^{\sigma}(t^{\tau\Psi}, \phi) = vars(R) \ \cup \ S$.
S4 For each $\rho \in \mathcal{N}(\tau)$, $\mathcal{E}^{\rho}(t, \tau) = vars(A) \ \cup \ B^{\rho}$.
S5 For each $\rho \in \mathcal{N}(\tau)$ with $\rho\Psi \bowtie \phi$, $\mathcal{Z}(\mathcal{E}^{\rho}(t, \tau)^{\rho\Psi}, \phi) = false$ iff $vars(C^{\rho} \cup A) \neq \emptyset$.
S6 For each $\rho \in \mathcal{N}(\tau)$ with $\rho\Psi \bowtie \phi$, $\mathcal{E}^{\sigma}(\mathcal{E}^{\rho}(t, \tau)^{\rho\Psi}, \phi) = vars(A) \cup vars(C^{\rho}) \cup D^{\rho}$.

S1 and S2 follow from Def. 4.5 and the definitions of R and A. S3 and S4 follow from Def. 4.6 and the definitions of R, S, A and B^{ρ}. S5 and S6 are proved below. First we prove S5.

$$\begin{aligned}
&\mathcal{Z}(\mathcal{E}^{\rho}(t, \tau)^{\rho\Psi}, \phi) = false \iff \\
&&\text{(by S4)}\\
&\mathcal{Z}((vars(A) \cup B^{\rho})^{\rho\Psi}, \phi) = false \iff \\
&&\text{(by Def. 4.5)}\\
&vars(\{r^{\omega} \mid r^{\omega} \text{ is a } \phi\text{−recursive subterm of } s^{\rho\Psi}, s \in vars(A) \cup B^{\rho}\}) \neq \emptyset \iff \\
&&\text{(by Def. 4.4)}\\
&vars(A) \cup vars(\{r^{\omega} \mid r^{\omega} \text{ is a } \phi\text{−recursive subterm of } s^{\rho\Psi}, s \in B^{\rho}\}) \neq \emptyset \iff \\
&&\text{(by Def. of } C^{\rho})\\
&vars(A) \cup vars(C^{\rho}) \neq \emptyset.
\end{aligned}$$

We now prove S6.

$$\begin{aligned}
&\mathcal{E}^{\sigma}(\mathcal{E}^{\rho}(t, \tau)^{\rho\Psi}, \phi) = \\
&&\text{(by S4)}\\
&\mathcal{E}^{\sigma}((vars(A) \cup B^{\rho})^{\rho\Psi}, \phi) = \\
&&\text{(by Def. 4.6)}\\
&vars(\{r^{\omega} \mid r^{\omega} \text{ is a } \phi\text{−recursive subterm of } s^{\rho\Psi}, s \in vars(A) \cup B^{\rho}\}) \cup \\
&\qquad \{r_i \mid f(\bar{r})^{\tau'\Psi'} \text{ is a } \phi\text{−recursive subterm of } s^{\rho\Psi}, s \in B^{\rho}, \tau'_i\Psi' = \sigma\} = \\
&&\text{(by Def. 4.4)}\\
&vars(A) \cup vars(\{r^{\omega} \mid r^{\omega} \text{ is a } \phi\text{−recursive subterm of } s^{\rho\Psi}, s \in B^{\rho}\}) \cup \\
&\qquad \{r_i \mid f(\bar{r})^{\tau'\Psi'} \text{ is a } \phi\text{−recursive subterm of } s^{\rho\Psi}, s \in B^{\rho}, \tau'_i\Psi' = \sigma\} = \\
&&\text{(by Def. of } C^{\rho}, D^{\rho})\\
&vars(A) \cup vars(C^{\rho}) \cup D^{\rho}.
\end{aligned}$$

Part 2: Let r^{ω} be a subterm of t^{τ} at depth d. We show by induction on d that $r^{\omega\Psi} \in R$ if and only if $r^{\omega} \in A$ or $r^{\omega\Psi} \in C^{\rho}$ for some $\rho \in \mathcal{N}(\tau)$ with $\rho\Psi \bowtie \phi$. For $d = 0$ this follows from the definitions of R and A.

Suppose now that r^{ω} is a subterm of t^{τ} at depth $d > 0$. Then there exists a subterm $f(\bar{r})^{\tau'\Psi'}$ of t^{τ} at depth $d - 1$ such that for some $i \in \{1, \ldots, n\}$, $r = r_i$ and $\omega = \tau'_i\Psi'$.
"$\Rightarrow$": Assume that $r^{\omega\Psi} \in R$. Since $\omega\Psi \bowtie \phi$, it follows from Lemma 4.1 that $\tau'\Psi'\Psi \bowtie \phi$ so that $f(\bar{r})^{\tau'\Psi'\Psi} \in R$. By the induction hypothesis there are two possibilities:

a) $f(\bar{r})^{\tau'\Psi'} \in A$. Since $\tau'\Psi' \bowtie \tau$, either $\omega \bowtie \tau$ or $\omega \in \mathcal{N}(\tau)$. If $\omega \bowtie \tau$ then $r^\omega \in A$. If $\omega \in \mathcal{N}(\tau)$, that is $\omega \in \mathcal{N}(\tau)$, then $r \in B^\omega$ and hence $r^{\omega\Psi} \in C^\omega$, and therefore $r^{\omega\Psi} \in C^\rho$ for some $\rho \in \mathcal{N}(\tau)$.
b) $f(\bar{r})^{\tau'\Psi'\Psi} \in C^\rho$ for some $\rho \in \mathcal{N}(\tau)$ with $\rho\Psi \bowtie \phi$. Since $\omega\Psi \bowtie \phi$ it follows that $r^{\omega\Psi} \in C^\rho$.

"$\Leftarrow$": Again we break this up into cases:

a) $r^\omega \in A$. Since $\omega \bowtie \tau$, it follows by Lemma 4.1 that $\tau'\Psi' \bowtie \tau$ so that $f(\bar{r})^{\tau'\Psi'} \in A$. By the induction hypothesis $f(\bar{r})^{\tau'\Psi'\Psi} \in R$. Since $\omega \bowtie \tau$ and $\tau\Psi \bowtie \phi$, it follows by Lemma A.4 that $r^{\omega\Psi} \in R$.
b) $r^{\omega\Psi} \in C^\rho$ for some $\rho \in \mathcal{N}(\tau)$ with $\rho\Psi \bowtie \phi$. By definition of C^ρ there are two possibilities: either $r \in B^\rho$, in which case $\omega = \rho$ and $f(\bar{r})^{\tau'\Psi'} \in A$, or $\omega\Psi \bowtie \phi$ and $f(\bar{r})^{\tau'\Psi'\Psi}$ is a subterm of an element of B^ρ. In the latter case, by Lemma 4.1, $\tau'\Psi'\Psi \bowtie \phi$ so that $f(\bar{r})^{\tau'\Psi'\Psi} \in C^\rho$.
In both cases, by the induction hypothesis $f(\bar{r})^{\tau'\Psi'\Psi} \in R$. In the first case, since $\omega = \rho$ and $\rho\Psi \bowtie \phi$, it follows that $r^{\omega\Psi} \in R$. In the second case, since $\omega\Psi \bowtie \phi$, $r^{\omega\Psi} \in R$.

Part 3: We prove (1). By S1, $\mathcal{Z}(t^{\tau\Psi}, \phi) = false$ if and only if $vars(R) \neq \emptyset$. By Part 2, $vars(R) \neq \emptyset$ if and only if $vars(A) \neq \emptyset$ or $vars(C^\rho) \neq \emptyset$ for some $\rho \in \mathcal{N}(\tau)$ with $\rho\Psi \bowtie \phi$. Then, by S2 and S5, this holds if and only if

$$\mathcal{Z}(t,\tau) \wedge \bigwedge_{\substack{\rho \in \mathcal{N}(\phi) \\ \rho\Psi \bowtie \phi}} \mathcal{Z}(\mathcal{E}^\rho(t,\tau)^{\rho\Psi}, \phi) = false.$$

Part 4: We prove (2) by showing that:

$$vars(R) \cup S = \bigcup_{\rho\Psi = \sigma} (vars(A) \cup B^\rho) \cup \bigcup_{\rho\Psi \bowtie \phi} (vars(C^\rho) \cup D^\rho).$$

The result then follows from S3, S4, and S6.

"$\subseteq$": For a variable $x \in R$ it follows by Part 2 that $x \in A$, or $x \in C^\rho$ for some $\rho \in \mathcal{N}(\tau)$ with $\rho\Psi \bowtie \phi$. For a term $r \in S$, there is $f(\bar{r})^{\tau'\Psi'\Psi} \in R$ such that $r = r_i$, and $\tau_i'\Psi'\Psi = \sigma$. By Part 2, either $f(\bar{r})^{\tau'\Psi'} \in A$, or $f(\bar{r})^{\tau'\Psi'\Psi} \in C^\rho$ for some $\rho \in \mathcal{N}(\tau)$ with $\rho\Psi \bowtie \phi$.

Assume first $f(\bar{r})^{\tau'\Psi'} \in A$. We show that $r \in B^\rho$ for some $\rho \in \mathcal{N}(\tau)$ with $\rho\Psi = \sigma$, namely $\rho = \tau_i'\Psi'$. Since by construction of A, $\tau_i'\Psi' \lhd^* \tau$, we only have to show that *not* $\tau_i'\Psi' \bowtie \tau$. By Lemma A.4, $\tau_i'\Psi' \bowtie \tau$, together with $\tau\Psi \bowtie \phi$, would imply $\tau_i'\Psi'\Psi \bowtie \phi$. This however is a contradiction, since it follows from $\tau_i'\Psi'\Psi = \sigma$ that $\tau_i'\Psi'\Psi \in \mathcal{N}(\phi)$.

Assume now $f(\bar{r})^{\tau'\Psi'\Psi} \in C^\rho$ for some $\rho \in \mathcal{N}(\tau)$ with $\rho\Psi \bowtie \phi$. Since $\tau_i'\Psi'\Psi = \sigma$ it follows that $r \in D^\rho$.

"$\supseteq$": For a variable $x \in A$, or $x \in C^\rho$ for some $\rho \in \mathcal{N}(\tau)$ with $\rho\Psi \bowtie \phi$, it follows by Part 2 that $x \in R$.

Secondly assume $r \in B^{\rho}$ for some $\rho \in \mathcal{N}(\tau)$ with $\rho\Psi = \sigma$. By definition, there is $f(\bar{r})^{\tau'\Psi'} \in A$ such that $r = r_i$ and $\tau_i'\Psi' = \rho$. By Part 2, $f(\bar{r})^{\tau'\Psi'\Psi} \in R$, and since $\tau_i'\Psi'\Psi = \sigma$, it follows that $r \in S$.

Thirdly assume $r \in D^{\rho}$ for some $\rho \in \mathcal{N}(\tau)$ with $\rho\Psi \bowtie \phi$. By definition, there is $f(\bar{r})^{\tau'\Psi'\Psi} \in C^{\rho}$ such that $r = r_i$ and $\tau_i'\Psi'\Psi = \sigma$. By Part 2, $f(\bar{r})^{\tau'\Psi'\Psi} \in R$, and since $\tau_i'\Psi'\Psi = \sigma$, it follows that $r \in S$. □

To prove Thm. 5.1, we need the following auxiliary lemma.

Lemma A.5. Let t^{τ} be a term. Every subterm of t^{τ} is either a recursive subterm of t^{τ}, or a subterm of a term in $\mathcal{E}^{\sigma}(t,\tau)$, for some $\sigma \in \mathcal{N}(\tau)$.

Proof. The proof is by induction on the depth of subterms of t^{τ}. For the base case observe that t^{τ} is a recursive subterm of itself.

Now suppose the result holds for all subterms of t^{τ} up to depth i. Let r^{ρ} be a subterm of t^{τ} at depth i and $w^{\omega} \lhd r^{\rho}$. If r^{ρ} is not a recursive subterm of t^{τ}, then r^{ρ} is a subterm of a term in $\mathcal{E}^{\sigma}(t,\tau)$ for some $\sigma \in \mathcal{N}(\tau)$, and thus w^{ω} is also a subterm of a term in $\mathcal{E}^{\sigma}(t,\tau)$. If r^{ρ} is a recursive subterm of t^{τ}, then since $\rho \bowtie \tau$ and $\omega \lhd \rho$, by Def. 4.2 either $\omega \bowtie \tau$ or $\omega \in \mathcal{N}(\tau)$. Thus either w^{ω} is a recursive subterm of t^{τ} or $w \in \mathcal{E}^{\omega}(t,\tau)$. □

Theorem 5.1. Let S be a set of terms having the same type. Then a variable occurs in an element of S (that is S is non-ground) if and only if `Any` or `Open` occurs in $\alpha(S)$.

Proof. There are three cases depending on whether S is empty, contains a variable, or neither.
Case 1: S is empty. Then $\alpha(S) = \texttt{Bot}$.
Case 2: S contains a variable x. Then $\alpha(x) = \texttt{Any}$ and thus $\alpha(S) = \texttt{Any}$.
Case 3: S contains no variables but contains a non-variable term. Then the type of terms in S is of the form $\tau\Psi$ for some type substitution Ψ and simple type $\tau = C(\bar{u})$. Suppose that $\mathcal{N}(\tau) = \langle \sigma_1, \dots, \sigma_m \rangle$ for some $m \geq 0$. Then there are abstract terms $b_1, \dots, b_m$ and termination flag b, such that

$$\alpha(S) = C^{\mathcal{A}}(b_1, \dots, b_m, b).$$

There are two subcases.
Case 3a: For some $t \in S$ and variable x, x^{ρ} is a recursive subterm of t^{τ}. Then $\mathcal{Z}(t,\tau) = \texttt{Open}$. Hence $b = \texttt{Open}$ and

$$\alpha(S) = C^{\mathcal{A}}(b_1, \dots, b_m, \texttt{Open}).$$

Case 3b: No term in S has a recursive subterm that is a variable. Then $\mathcal{Z}(t,\tau) = \texttt{Ter}$ for each $t \in S$. Hence, by Def. 5.2, $b = \texttt{Ter}$. The proof for this case is by induction on the length of the longest NRS-sequence (see Lemma A.2) for $\tau\Psi$. The base case is when $m = 0$. Then by Lemma A.5, every term in S is ground and $\alpha(S) = C^{\mathcal{A}}(\texttt{Ter})$.

Now suppose $m > 0$. By Lemma A.5, S contains a non-ground term if and only if $\mathcal{E}^{\sigma_j}(t, \tau)$ contains a non-ground term for some $t \in S$ and $j \in \{1, \dots, m\}$. By Def. 5.3

$$\alpha(S) = \sqcup\{C^{\mathcal{A}}(\alpha(\mathcal{E}^{\sigma_1}(t, \tau)), \dots, \alpha(\mathcal{E}^{\sigma_m}(t, \tau)), \mathtt{Ter}) \mid t^{\tau} \in S\}.$$

Thus, by Def. 5.2 and Def. 5.3, for each $j \in \{1, \dots, m\}$, $b_j = \alpha(\mathcal{E}^{\sigma_j}(S, \tau))$. Let $j \in \{1, \dots, m\}$. If $\mathcal{E}^{\sigma_j}(S, \tau)$ is empty, by case 1 above, $\alpha(\mathcal{E}^{\sigma_j}(S, \tau)) = \mathtt{Bot}$. If $\mathcal{E}^{\sigma_j}(S, \tau)$ contains a variable, by case 2 above, $\alpha(\mathcal{E}^{\sigma_j}(S, \tau)) = \mathtt{Any}$. Otherwise, $\mathcal{E}^{\sigma_j}(S, \tau)$ contains a non-variable term and the terms in $\mathcal{E}^{\sigma_j}(S, \tau)$ have type $\sigma_j \Psi$, for which, by induction hypothesis, the result holds. Hence b_j has an occurrence of `Any` or `Open` if and only if $\mathcal{E}^{\sigma_j}(S, \tau)$ contains a non-ground term. It follows that $\alpha(S)$ has an occurrence of `Any` or `Open` if and only if S contains a non-ground term. $\square$