Mode Analysis Domains for Typed Logic Programs

Jan–Georg Smaus* Pat Hill Andy King

Extended abstract

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 on the expressiveness of the abstract domain. This work [3] 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. The domain construction procedure is fully implemented using the Gödel language and tested on a number of example programs.

1 Introduction

Types are used in programming to restrict the underlying syntax so that only meaningful expressions are allowed. Then many programming errors can be detected by the compiler.

Modes characterise the degree to which program variables are instantiated at certain program points. 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. Our notion of mode is a *descriptive* one: we do not assume that modes are enforced by the compiler.

This abstract describes 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. These domains characterise varying degrees of instantiation of terms of the corresponding type.

The procedure for constructing such domains is implemented (in Gödel) for Gödel programs to demonstrate that an analysis using our domains is viable and produces precise results.

This work is the natural generalisation of [1] and takes the idea presented there to its limits by providing the highest degree of precision a generic domain construction should provide.

2 Motivating and Illustrative Examples

In the examples we use Gödel syntax [2]. Variables and (type) parameters begin with lower case letters; other alphabetic symbols begin with upper case letters.

Example 2.1 This is the usual list type. It is also considered in [1].

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

`List` is a (type) constructor; `u` is a type parameter that can be instantiated to any type such as `Int` or `List(Int)`; `Nil` is a constant of type `List(u)`; and `Cons` is the usual list constructor. It is common to distinguish *nil-terminated* lists from *open* lists.

*University of Kent at Canterbury, j.g.smaus@ukc.ac.uk, phone xx44/1227/827553, fax xx44/1227/762811.

Example 2.2 This example formalises nested lists by the type `Nest(v)`. The notable property of the declaration for `N` is that the range type, `Nest(v)`, is a proper *subexpression* of the argument type `List(Nest(v))`. However this occurs in "real" programs, and it cannot be dealt with in [1].

```
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. `Tables` is implemented in Gödel as an AVL-tree. This example can also not be dealt with in [1].

```
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).
```

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 wrt. $\prec$ is denoted by $\bar{u}$.

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 $f_{\langle\tau_1 \dots \tau_n, \tau\rangle} \in \Sigma_f$ (resp. $p_{\langle\tau_1 \dots \tau_n\rangle} \in \Sigma_p$) then $\langle\tau_1, \dots, \tau_n\rangle \in T(\Sigma_\tau, U)^\star$ and $\tau \in T(\Sigma_\tau, U) \setminus U$. We call τ the **range type** of $f_{\langle\tau_1 \dots \tau_n, \tau\rangle}$. Terms and atoms are defined in the usual way [2]. 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 variables, then the triple $L = \langle\Sigma_p, \Sigma_f, V\rangle$ defines a **polymorphic many-sorted first order language**. Tuples of *distinct* variables are denoted by $\bar{y}$. The set of variables in o is denoted by $vars(o)$.

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

4 The Structure of Terms and Types

In this section we formalise *which* aspects of the structure of a term should be characterised by an abstract term. We describe how function declarations relate types to one another. We then define *termination* of a term, as well as functions which extract certain subterms of a term.

4.1 Relations between Types

Definition 4.1 A **simple type** is a type of the form $C(\bar{u})$, where $C \in \Sigma_\tau$.

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

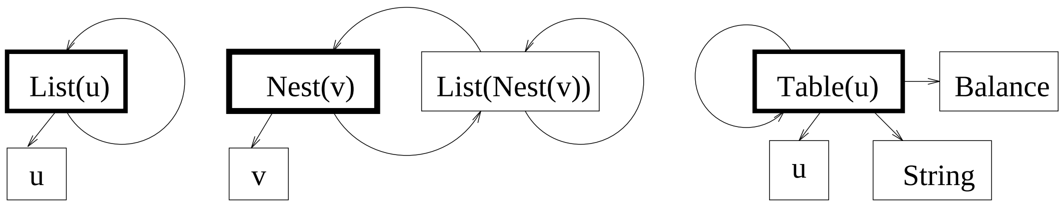

Figure 1: Some type graphs, with initial node highlighted

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.

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

We define $\mathcal{N}(\phi)$ as the set of types σ such that $\phi \not\lhd^* \sigma$ and there exists τ such that $\sigma \lhd \tau$ and $\tau \bowtie \phi$. We call $\mathcal{N}(\phi)$ the set of **non-recursive subterm types of** ϕ. 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 denote the *tuple* $\langle \sigma_1, \ldots, \sigma_m \rangle$ by $\mathcal{N}(\phi)$ as well.

Note that $\tau \bowtie \phi$ includes the case that $\tau = \phi$, and that if $\sigma \in \mathcal{N}(\phi)$, then $\sigma \not\bowtie \phi$. The relation $\lhd$ can be visualised as a *type graph* [5]. 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 iff $\sigma_2 \lhd \sigma_1$. The recursive types of ϕ are all the types in the strongly connected component (SCC) of ϕ, and $\mathcal{N}(\phi)$ contains the types σ such that there is an edge from the SCC of ϕ to σ.

Example 4.1 In Fig. 1 there is a type graph for each of the examples in Sect. 2. Trivially `Int` $\bowtie$ `Int`. However, `List(u)` $\bowtie$ `List(u)` is non-trivial in that, in the type graph for `List(u)`, there is a path from `List(u)` to itself. Furthermore `List(Nest(v))` $\bowtie$ `Nest(v)`.

4.2 Traversing Concrete Terms

A term t is annotated with a type ϕ by writing t^ϕ. This 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 4.2 Let t^ϕ be a term. Then t^ϕ is a **subterm of** t^ϕ at **depth** 0. If $s = f_{\langle \tau_1 \ldots \tau_n, \tau \rangle}(s_1, \ldots, s_n)$ and for some type substitution ψ, $s^{\tau\psi}$ is a subterm of t^ϕ at depth d, then $s_i^{\tau_i\psi}$ is a **subterm of** t^ϕ at **depth** $d+1$ for $i \in \{1, \ldots, n\}$. We write $s^\sigma \lhd^* t^\phi$ if s^σ is a subterm of t^ϕ at some depth d ($s^\sigma \lhd t^\phi$ when $d = 1$).

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

Example 4.2 $\mathtt{x}^{\mathtt{v}}$ 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 4.3 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.

The correspondence between subterms and subterm types can be illustrated by drawing the term as tree that resembles the corresponding type graph.

Example 4.3
The term tree for $t = \mathtt{N}([\mathtt{E}(7)])^{\mathtt{Nest(v)}}$ is given in Fig. 2 where the node for t is highlighted. Each box drawn with solid lines stands for a subterm. This tree can be mapped 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 $7^{\mathtt{v}}$ are recursive subterms of t. 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. 4.3, take $\sigma = \phi = \mathtt{Nest(v)}$ and $\tau = \mathtt{List(Nest(v))}$). However, $\mathtt{E}(7)^{\mathtt{u}}$ is not a recursive subterm of $[\mathtt{E}(7)]^{\mathtt{List(u)}}$.

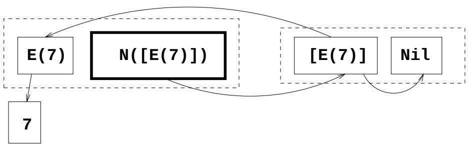

Figure 2: Term tree for $\mathtt{N}([\mathtt{E}(7)])^{\mathtt{Nest(v)}}$

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

Definition 4.4 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.

A term t is **terminated** if $t = f_{\langle \tau_1 \ldots \tau_n, \tau \rangle}(t_1, \ldots, t_n)$ and $\mathcal{Z}(t^\tau, \tau) = \mathit{true}$. A term is **open** if it is not terminated. 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 $\tau = \phi$.

Example 4.4 Any variable $\mathtt{x}$ 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 $\mathtt{Nest(v)}$-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.

Definition 4.5 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 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 4.5 For the term $\mathtt{N}([\mathtt{E}(7)])$, we have $\mathcal{E}^{\mathtt{v}}(\mathtt{N}([\mathtt{E}(7)]), \mathtt{Nest(v)}) = \{7\}$. $\mathcal{N}(\mathtt{Table(u)}) = \{\mathtt{u}, \mathtt{Balance}, \mathtt{String}\}$, and so there are three extractor functions: $\mathcal{E}^{\mathtt{u}}$, which extracts all value subterms; $\mathcal{E}^{\mathtt{Balance}}$, which extracts the subterms containing balancing information; and $\mathcal{E}^{\mathtt{String}}$, which extracts all key subterms.

Note 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.

5 Abstract Terms and Abstract Programs

5.1 Abstraction of Terms

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

Definition 5.1 If ϕ is a parameter, define $\mathcal{D}_\phi = \{\mathtt{Bot}, \mathtt{Any}\}$. 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

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

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

The flag `Ter` abstracts the property of a term being terminated and `Open` that of being open. Note that for some types, for example `Int`, a term can be open only if it is a variable.

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

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

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

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

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

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

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

Definition 5.2 For the termination flags define $\mathtt{Ter} < \mathtt{Open}$. For abstract terms, $<$ is defined as follows:

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

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, to formalise the relationship between concrete and abstract terms.

Definition 5.3 Let $\tau = C(\bar{u})$ and $\mathcal{N}(\tau) = \langle \sigma_1, \ldots, \sigma_m \rangle$. For the truth values define $\alpha(true) = \mathtt{Ter}$ and $\alpha(false) = \mathtt{Open}$. If S is a set of terms, define $\alpha(S) = \sqcup\{\alpha(t) \mid t \in S\}$, where $\alpha(t)$ is defined as:

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

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

Example 5.2 We give some terms and their abstractions.

term	abstraction
`x`	`Any`
`7`	$\mathtt{Int}^{\mathcal{A}}(\mathtt{Ter})$
`[]`	$\mathtt{List}^{\mathcal{A}}(\mathtt{Bot}, \mathtt{Ter})$
`[7,x]`	$\mathtt{List}^{\mathcal{A}}(\mathtt{Any}, \mathtt{Ter})$
`[7|x]`	$\mathtt{List}^{\mathcal{A}}(\mathtt{Any}, \mathtt{Open})$

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

5.2 Traversing Abstract Terms

In order to define abstract unification, we require an abstract termination function and abstract extractors similar to those already defined for concrete terms.

Definition 5.4 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 from a domain for an instance of $\tau\psi$.

1. Abstract termination function.

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

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

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

We omit the annotation $\tau\psi$ in the expressions $\mathcal{AZ}(b^{\tau\psi}, \phi)$ and $\mathcal{AE}^{\sigma}(b^{\tau\psi}, \phi)$ whenever $\tau = \phi$ and ψ is the identity. In this case, the abstract termination function is merely a *projection* onto the termination flag of the abstract term. Similarly, the abstract extractor for σ is merely a projection onto an argument of the abstract term.

Example 5.3

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

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

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

Theorem 5.1 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

$$\alpha(\mathcal{Z}(t^{\tau\psi}, \phi)) \;=\; \mathcal{AZ}(\alpha(t)^{\tau\psi}, \phi) \qquad (1)$$

$$\alpha(\mathcal{E}^{\sigma}(t^{\tau\psi}, \phi)) \;=\; \mathcal{AE}^{\sigma}(\alpha(t)^{\tau\psi}, \phi) \qquad (2)$$

5.3 Abstract Compilation

We define an abstract program and show that it is a safe approximation of the concrete program wrt. 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, i.e. for each $f \in \Sigma_f$, a predicate which expresses the dependency between $\alpha(f(t_1, \ldots, t_n))$ and $\alpha(t_1), \ldots, \alpha(t_n)$.

Definition 5.5 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 $f_{\mathrm{dep}}(C^{\mathcal{A}}(a_1, \ldots, a_m, c), b_1, \ldots, b_n)$ **holds** if

$$a_j \;=\; \sqcup(\{b_i \mid \tau_i = \sigma_j\} \cup \{\mathcal{AE}^{\sigma_j}(b_i^{\tau_i}, \tau) \mid \tau_i \bowtie \tau\}) \qquad \text{for all } j \in \{1, \ldots, m\} \qquad (1)$$

$$c \;=\; \bigwedge_{\tau_i \bowtie \tau} \mathcal{AZ}(b_i^{\tau_i}, \tau) \qquad (2)$$

Example 5.4 To give an idea of how Def. 5.5 translates into code, consider `Cons`. Assuming that $\texttt{Lub}(a,b,c)$ 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).
```

Definition 5.6 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 $K = h \leftarrow g_1 \wedge \ldots \wedge g_l$ define

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

For a program P whose set of clauses is S 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 5.5 We give the usual recursive clause for `Append` 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 program 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.

Definition 5.7 Let P be a program. The relation $\xrightarrow{P}$ between states is defined by the following rules:

$$\begin{array}{llll} \langle h_1 : \ldots : h_l, \theta \rangle \xrightarrow{P} & \langle h_2 : \ldots : h_l, \theta\theta' \rangle & \text{if } h_1 \text{ is `} x = t \text{' and } x\theta\theta' = t\theta\theta' & (1) \\ \langle h_1 : \ldots : h_l, \theta \rangle \xrightarrow{P} & \langle G : h_2 : \ldots : h_l, \theta\theta' \rangle & \text{if } h \leftarrow G \in_{\approx} S \text{ and } h\theta\theta' = h_1\theta\theta' & (2) \end{array}$$

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

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

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.

Theorem 5.2 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 $\theta^\alpha = \{x/\alpha(x\theta) \mid x \in \mathit{dom}(\theta)\}$.

6 Implementation and Results

We have implemented our mode analysis for object programs in Gödel and analysed several small to medium size typed programs. All call and answer patterns, which may arise in a derivation of an abstract program for a given query, are computed in this analysis. By Theorem 5.2, these patterns correspond to patterns in the derivation of the concrete program.

For example, we analysed the `Insert`/5 predicate from the `Tables` module (see Ex. 5.1). The arguments 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 is v. With our abstract domain for Tables(Int), we can deduce that a call pattern[2]

$$\texttt{Insert}(\texttt{Tab}^{\mathcal{A}}(\texttt{Int}^{\mathcal{A}},\texttt{Bal}^{\mathcal{A}},\texttt{Str}^{\mathcal{A}},\texttt{Ter}),\texttt{Str}^{\mathcal{A}},\texttt{Any},\texttt{Any})$$

results in the answer pattern

$$\texttt{Insert}(\texttt{Tab}^{\mathcal{A}}(\texttt{Int}^{\mathcal{A}},\texttt{Bal}^{\mathcal{A}},\texttt{Str}^{\mathcal{A}},\texttt{Ter}),\texttt{Str}^{\mathcal{A}},\texttt{Any},\texttt{Tab}^{\mathcal{A}}(\texttt{Any},\texttt{Bal}^{\mathcal{A}},\texttt{Str}^{\mathcal{A}},\texttt{Ter})).$$

So inserting an uninstantiated value into a ground table results in a table whose values may be uninstantiated. Similarly, we could infer precise modes for the other programs we analysed.

7 Discussion and Related Work

For space reasons, we can only compare our work to three other approaches.

We have presented a general domain construction for mode analysis of typed logic programs. For common examples (lists, binary trees), our formalism is simple and yields abstract domains that are comparable to [1]. However, our work is a non-trivial generalisation of ideas presented in [1] for untyped languages. There, all types are essentially lists and trees. The complexity of our formalism is primarily due to function declarations where the range type occurs again as a *proper* subexpression of an argument type, such as the declaration of N in Ex. 2.2. Apart from the fact that such types occur in "real" programs, our work is a contribution because it helps to understand other, more ad-hoc domain constructions as instances of a general theory.

A complex system for type analysis of Prolog is presented in [5]. 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. In contrast, we exploit the type declarations to construct domains that are inherently finite.

Mercury [4] has a mode system based on *instantiation states*, which 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 is comparable in precision to an abstract term in our formalism. In Mercury, it is the *user* who has to specify a set of instantiation states by declaring the mode. In contrast, we have described how the abstract terms and their values can be inferred automatically.

Acknowledgements We thank Tony Bowers, Henning Christiansen, Bart Demoen, Andrew Heaton, Fergus Henderson, Jonathan Martin and Lambert Meertens for helpful discussions.

References

[1] M. Codish and V. Lagoon. Type dependencies for logic programs using ACI-unification. In *Proceedings of the Israeli Symposium on Theory of Computing and Systems*, pages 136–145. IEEE Press, 1996.

[2] P. M. Hill and J. W. Lloyd. *The Gödel Programming Language*. MIT Press, 1994.

[3] J.-G. Smaus, P. M. Hill, and A. M. King. Domain construction for mode analysis of typed logic programs. Technical Report 8-97, University of Kent at Canterbury, 1997.

[4] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *JLP*, November 1996.

[5] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of Prolog using type graphs. *JLP*, 22(3):179–209, 1995.

[2] We use some abbreviations and omit the termination flag for types Int, Balance and String.