Domain Construction for Mode Analysis of Typed Logic Programs

Jan–Georg Smaus*, Pat Hill† and Andy King*

March 1997

Abstract

There are many applications where precise mode analysis is required. However, within the framework of abstract interpretation, the precision of an analyser 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 where each non-variable symbol is explicitly typed. We show how to construct precise domains and their abstraction functions that reflect the declared structure of terms. This domain construction is modular in that an abstract domain for a type does not depend on modules that import this type. A program is abstracted by replacing the unification operations with abstract unification operations. The precision of the domains is demonstrated for several examples using Gödel programs. Correctness of the method is proven. The domain construction has been implemented in Gödel.

Keywords: Logic programming, types, abstract interpretation, mode analysis, polymorphism, abstract domains, Gödel.

1 Introduction

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.

Types restrict the underlying syntax so that only meaningful expressions are allowed. This enables most typographical errors and inconsistencies in the knowledge representation to be detected by the compiler. As a consequence, an increasing number of applications using typed languages such as Mercury [14] or Gödel [8] are being developed.

In this paper we present new abstract domains for determining mode and related information in typed logic programs. The explicitly declared type information provided by the typed programs is used to determine, for each type, a specialised abstract domain and a definition of abstract unification.

Such specialised domains are useful for characterising the behaviour of programs that may operate on partially instantiated data-structures. This has been demonstrated in [3] for lists.

*Computing Laboratory, University of Kent at Canterbury, Canterbury, CT2 7NF, United Kingdom. Tel. xx44/1227/827553. Fax xx44/1227/762811. {j.g.smaus, a.m.king}@ukc.ac.uk

†School of Computer Studies, University of Leeds, Leeds, LS2 9JT, United Kingdom. hill@scs.leeds.ac.uk

Rather than only distinguishing between ground and non-ground terms, [3] introduces a domain which can characterise a nil-terminated list. The domain can also characterise, for such a list, the instantiation of the list elements. We propose a domain for lists which is even more precise in that it can distinguish a non-variable *not* nil-terminated list from an unbound variable. Characterising the property of being *non-variable* is particularly important for programs that employ co-routining by means of delay declarations. Moreover, if the analyser only allows the more expressive domains for a fixed number of "standard" types such as a list or binary tree (as in [3]), then any non-standard type (e.g. a ternary tree or a tree with an extra argument that represents balancing information) in the analysed program may not be analysed with the precision afforded to the standard types. Our abstract domains can be generated automatically so that analysers incorporating such a domain generator will be robust with respect to such changes in the data structures.

The rest of the paper is organised as follows. Section 2 illustrates our domain construction with an example. Section 3 gives a brief overview of polymorphic many-sorted languages and programs, defining much of the syntax used in the rest of the paper. Section 4 introduces the basic concepts for terms and types that are needed for the definition of the abstract domains. Section 5 defines the abstract domains and the abstraction functions for terms and programs. The relationship between concrete and abstract programs is formalised. Section 6 reports on the implementation. Section 7 concludes by discussing related and future work.

2 Motivating Example

The syntax used here is that of Gödel [8]. Gödel is a typed language and, as it has a distinctive syntax, confusion with the (untyped) language Prolog is avoided. Variables and parameters begin with lower case letters; constants, functions, (type) bases, (type) constructors, and predicates begin with upper case letters.

We use an example invented by Tony Bowers [2] to disprove a common point of criticism that "list flattening" cannot be realised in Gödel, i.e. terms such as `[1,[2,3]]` cannot be defined, let alone flattened. We do not know of any attempt to analyse the Prolog equivalent [13] of this program using fixed domains for "standard" types. The `Nests` module provides a clear concept of nested lists, formalised by the type `Nest(u)`. A trivial nest is constructed using function `E`, a complex nest by "nesting" a list of nests using function `N`. The code for flattening a nest "writes itself by case analysis" [2]. The declaration for `N` is unusual in that the range type, `Nest(u)`, is a proper subterm of the argument type `List(Nest(u))`. This is clearly a non-trivial example of polymorphism. The program is as follows:

```
MODULE          Nests.
IMPORT          Lists, Integers.
CONSTRUCTOR     Nest/1.
FUNCTION        E : u -> Nest(u);
                N : List(Nest(u)) -> Nest(u).
PREDICATE       Flatten : Nest(u) * List(u);
                FlattenAcc : Nest(u) * List(u) * List(u);
                FlattenListOfNests : List(Nest(u)) * List(u) * List(u).
Flatten(nest, list) <-
   FlattenAcc(nest, list, []).

FlattenAcc(E(x), [x|list_t], list_t).
FlattenAcc(N(x), list, list_t) <-
   FlattenListOfNests(x, list, list_t).
```

```
FlattenListOfNests([], list, list).
FlattenListOfNests([x|xs], list, list_t) <-
   FlattenAcc(x, list, list_1)  &  FlattenListOfNests(xs, list_1, list_t).
```

It may be surprising that in the last clause, `FlattenAcc` is called before `FlattenListOfNests`, but the accumulator `list_t` is passed to `FlattenListOfNests` rather than `FlattenAcc`. The reason for this is that it keeps the list elements of the first argument of `Flatten` in the right order, as well as preventing looping.

2.1 Abstract domains

The structure of the abstract domains is determined by the type structure of the module. To illustrate the abstract domain for a simple type containing only constants, we use the base type `Integer`[1] (abbreviated as `Int`). This means that a variable `x` of type `Int` can only be free or ground. If `x` cannot be inferred to be ground, then, regardless of its type, it is characterised by `Any`. If `x` is known to be ground, it is characterised by the *abstract term* `Int_a`.

The type `List(Int)` is more complex than `Int`. A term of type `List(Int)` can be free, a not nil-terminated list, or a nil-terminated list. The following table gives some examples of terms of type `List(Int)` and corresponding abstract terms, where `Ter` stands for "terminated".

term	abstraction	term	abstraction
`x`	`Any`	`[x]`	`List_a(Any,Ter)`
`[1|y]`	`List_a(Any,Open)`	`[1,2]`	`List_a(Int_a,Ter)`

Such abstract terms characterise nil-termination and, recursively, the instantiation of the list elements. The latter is approximated by the minimum degree of instantiation for *all* elements, ensuring the abstract domain is finite. Note that the list elements are only characterised for a nil-terminated list, for the characterisation to be downwards closed [12].

Using identifiers for the abstract terms whose names stem from the types does not serve the purpose of type inference as in the analysis of untyped programs, but is useful for various reasons, the simplest being that names such as `List_a` and `Int_a` associate the abstract term with the type of the term being abstracted. Note that the subscript `a` stands for "abstract". This example shows that, for lists, our abstract domain is similar to the domain proposed in [3] but even more expressive.

The principles used for the abstract domain for `List(Int)` can be generalised to `Nest(Int)` or any other type. Terms of type `Nest(Int)` can have a complicated structure. The following table gives the types and abstract terms for some terms in the `Nests` program.

term	type	abstraction
`E(7)`	`Nest(Int)`	`Nest_a(Int_a,Ter)`
`[E(7)]`	`List(Nest(Int))`	`List_a(Nest_a(Int_a,Ter),Ter)`
`N([E(7)])`	`Nest(Int)`	`Nest_a(Int_a,Ter)`
`N([E(7),y])`	`Nest(Int)`	`Nest_a(Any,Open)`
`N([E(7)|y])`	`Nest(Int)`	`Nest_a(Any,Open)`

The abstract term for a term t of type `Nest(Int)` characterises *termination* of t and the instantiation of the subterms of t of type `Int`. Here, *termination* generalises the well-known concept of nil-termination for lists and means that no instance of t has a variable subterm of type `Nest(Int)`. We define this property for arbitrary types later. The domain construction is modular in the sense that the domain for `Nest(Int)` depends on the module `Lists`, but the

[1] The Gödel module `Lists` exports the types `Integer` and `List(u)` with the usual notation for lists.

domain for `List(Int)` does not depend on the module `Nests`. Modularity is crucial because the abstraction of a term like `Cons(y,z)` should be independent of the context in which it occurs.

Since integers are assumed to be constants and trivially terminated, there is no termination argument in the abstract term for `Integer`. In the presentation of the theory however, we *always* reserve termination argument to save unnecessary case distinctions.

2.2 Tracing program execution

It is a common assumption in abstract interpretation [4, 5, 11] that programs are in normal form: All literals in a clause are of the form $Q(y_1, \ldots, y_l)$ or $x = f(y_1, \ldots, y_l)$ or $x = y$, where the y_k are all different variables. This means that all unifications in a program are made explicit by equations. There is no loss of generality since every program can be translated into normal form. However for readability, we have not presented the `Nests` example in normal form.

In order to trace the instantiation of variables in the `Nests` program, the program execution is mimicked using the *abstraction* of `Nests`, which is defined by replacing all unifications in the concrete program with unifications defined on abstract terms. E.g. a unification `x = Cons(y,z)`[2] is abstracted by the atom `Cons_dep(x,y,z)` which defines the dependency between the abstract term for `x`[3] and the abstract terms for `y` and `z`.

Mode analysis makes statements about the instantiation of the arguments of a predicate when it is *called* and when it is *answered*. This information is obtained by evaluating the call and answer patterns of the abstract program. Using any appropriate abstract interpretation framework the `Flatten` predicate has the following call and answer patterns:

Assumed initial call pattern	Inferred answer pattern
`Flatten(Nest_a(Int_a,Ter),Any)`	`Flatten(Nest_a(Int_a,Ter),List_a(Int_a,Ter))`
`Flatten(Nest_a(Any,Ter),Any)`	`Flatten(Nest_a(Any,Ter), List_a(Any,Ter))`
`Flatten(Any,List_a(Int_a,Ter))`	`Flatten(Nest_a(Int_a,Ter),List_a(Int_a,Ter))`

3 Preliminaries

Let Σ_τ (resp. Σ_f) be an alphabet of type constructor (resp. function) symbols which includes at least one base (resp. constant) and let Σ_p be an alphabet of predicate symbols. Let U be a countably infinite set of type parameters so that the term structure $T(\Sigma_\tau, U)$ is the set of parametric types. We define the order $\prec$ on types as the order induced by some (arbitrary but fixed) order on constructor and parameter symbols, where parameter symbols come before constructor symbols.

Let $V = \{V_\tau \mid \tau \in T(\Sigma_\tau, U)\}$ be a family of countably infinite, disjoint sets of variables, where each $x_\tau \in V_\tau$ has type τ. 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 $\tau_1, \ldots, \tau_n \in T(\Sigma_\tau, U)^\star$ and $\tau \in T(\Sigma_\tau, U) \setminus U$. We call τ the *range type* of $f_{\langle \tau_1 \ldots \tau_n, \tau \rangle} \in \Sigma_f$. A symbol will often be written without its type if it is clear from the context. Terms and atoms are defined in the usual way [8]. The triple $L = \langle \Sigma_p, \Sigma_f, V \rangle$ defines a polymorphic many-sorted first order language.

In the rest of the paper, u, v denote parameters; x, y, z variables; t terms; ρ, θ (term) substitutions; σ, τ, ϕ types; and ψ type substitutions. $\bar{x}, \bar{y}$ (resp. $\bar{u}, \bar{v}$) denote tuples of *distinct* variables (resp. parameters). $\bar{t}$ (resp. $\bar{\tau}$) denote tuples of terms (resp. types).

A *literal*[4] is either of the form $x = f(\bar{y})$, where $f \in \Sigma_f$, or of the form $x = y$, or of the form $Q(\bar{y})$, where $Q \in \Sigma_p$. A *statement* is of the form $Q(\bar{y}) \leftarrow g_1 \wedge \ldots \wedge g_l$ where $Q \in \Sigma_p$, $l \geq 0$ and

[2] For consistency with the notation for the abstract domain, `Cons(y,z)` rather than `[y|z]` is used here.
[3] More precisely: the abstract term for the concrete term to which `x` is bound.
[4] We ignore negated literals here, although we have considered them in the implementation.

each g_k is a literal. If S is a set of statements, then the tuple $P = \langle L, S\rangle$ defines a polymorphic many-sorted logic program.

4 The Structure of Terms and Types

An *abstract* term characterises the structure of a concrete term. Thus, to define the abstraction, we must first formalise the characteristics of the concrete structure that we wish to abstract.

4.1 Relations between types

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

Definition 4.2 [subterm type] Let $\sigma, \phi \in T(\Sigma_\tau, U)$. We say σ is a **direct subterm type of** ϕ (denoted by $\sigma \lhd \phi$) if there is $f_{\langle \tau_1 \ldots \tau_n, \tau\rangle} \in \Sigma_f$ and a type substitution ψ such that $\psi(\tau_i) = \sigma$ and $\psi(\tau) = \phi$ for some $i \in \{1, \ldots, n\}$. The transitive closure of $\lhd$ is denoted by $\lhd^+$ and the transitive, reflexive closure by $\lhd^*$.

We impose the following two restrictions on the declarations of the programs we consider:

Simple Range Condition: For all $f_{\langle \tau_1 \ldots \tau_n, \tau\rangle} \in \Sigma_f$, τ must be a simple type.
Reflexive Condition: If $\psi(\sigma) \lhd^+ \sigma$ for some type σ then ψ must be the identity substitution.

Example 4.1 The following is a program which violates these two conditions.

```
MODULE       StrangeLists.
IMPORT       Numbers.
CONSTRUCTOR  SList/1.
CONSTANT     Nil : SList(a).
FUNCTION     Cons : a * SList(a) -> SList(a);
             F : Float * Integer -> SList(Rational);
             G : SList(Integer) -> SList(a).
```

We do not know of any real cases where these conditions are violated [2], and not only do our proofs rely on these, but also allowing for such programs would complicate the presentation. In typed languages such as Mercury, the actual syntax ensures these conditions are always satisfied [7, 14]. Note that the `Nests` program satisfies both conditions.

Definition 4.3 [recursive type and non-recursive subterm type] Let ϕ be a simple type. We say σ is a **recursive type of** ϕ (denoted by $\sigma \trianglelefteq \phi$) if $\sigma \lhd^* \phi$ and $\phi \lhd^* \sigma$.

We say σ is a **non-recursive subterm type of** ϕ (denoted by $\sigma \lll \phi$) if $\phi \not\lhd^* \sigma$ and there is a τ such that $\sigma \lhd \tau$ and $\tau \trianglelefteq \phi$. We define $\mathcal{N}(\phi) = \{\sigma \mid \sigma \lll \phi\}$. If $\mathcal{N}(\phi) = \{\sigma_1, \ldots, \sigma_m\}$ and $\sigma_j \prec \sigma_{j+1}$ (see Section 3) for all $j \in \{1, \ldots, m-1\}$, we sometimes abuse notation and denote the tuple $\langle \sigma_1, \ldots, \sigma_m\rangle$ by $\mathcal{N}(\phi)$ as well.

Note that every type is a recursive type of itself. The non-trivial case is when $\sigma \trianglelefteq \phi$ and also $\sigma \lhd^+ \phi$. In the example of Section 2 and in the implementation we distinguish between the cases, but in the theory a uniform presentation is preferred.

Example 4.2 For `List(u)` defined as usual [8] we have `u` $\lll$ `List(u)` and `List(u)` $\trianglelefteq$ `List(u)`. For the `Nests` module of Section 2 we have

`u` $\lhd$ `List(u)` `u` $\lhd$ `Nest(u)` `List(u)` $\lhd$ `List(u)` `List(Nest(u))` $\lhd$ `Nest(u)`
`u` $\lll$ `List(u)` `u` $\lll$ `Nest(u)` `List(u)` $\trianglelefteq$ `List(u)` `List(Nest(u))` $\trianglelefteq$ `Nest(u)` `Nest(u)` $\trianglelefteq$ `Nest(u)`.

Example 4.3

```
BASE            Day.
CONSTRUCTOR     EvenList/1, OddList/1.
CONSTANT        Nil : EvenList(u);
                Monday, ..., Sunday : Day.
FUNCTION        EvenCons : u * OddList(u) -> EvenList(u);
                OddCons : u * EvenList(u) -> OddList(u).
```

This module defines two types for lists of even and odd numbers of elements and a base type `Day`. `EvenList(u)` and `OddList(u)` are recursive types of themselves (trivially) and of each other. $\mathcal{N}(\texttt{OddList(u)}) = \mathcal{N}(\texttt{EvenList(u)}) = \{\texttt{u}\}$ and $\mathcal{N}(\texttt{Day}) = \emptyset$.

The following lemmas are needed as auxiliary statements in later proofs.

Lemma 4.1 Let σ, ϕ be types and ψ be a type substitution. If $\sigma \lhd \phi$ then $\psi(\sigma) \lhd \psi(\phi)$. If $\sigma \lhd^+ \phi$ then $\psi(\sigma) \lhd^+ \psi(\phi)$. If $\sigma \lhd^* \phi$ then $\psi(\sigma) \lhd^* \psi(\phi)$.
Proof: For the first statement, there is an $f_{\langle \tau_1 \dots \tau_n, \tau \rangle} \in \Sigma_f$ and a type substitution ψ' such that for some $i \in \{1, \dots, n\}$, $\psi'(\tau_i) = \sigma$ and $\psi'(\tau) = \phi$. Consequently $\psi(\psi'(\tau_i)) = \psi(\sigma)$ and $\psi(\psi'(\tau) = \psi(\phi)$, so $\psi(\sigma) \lhd \psi(\phi)$. The other statements follow from the first. ∎

Lemma 4.2 Let $\sigma_1, \sigma_2, \sigma_3$ be types. If $\sigma_1 \trianglelefteq \sigma_2$ and $\psi(\sigma_2) \trianglelefteq \sigma_3$ for some type substitution ψ then $\psi(\sigma_1) \trianglelefteq \sigma_3$.
Proof: By Lemma 4.1 we have $\psi(\sigma_1) \lhd^+ \sigma_3$ and $\sigma_3 \lhd^* \psi(\sigma_1)$. Furthermore σ_3 is simple. ∎

Lemma 4.3 Let σ, ϕ be types. If $\sigma \mathrel{<\!\!\lhd} \phi$ then $\psi(\phi) \not\lhd^* \sigma$ for all type substitutions ψ.
Proof: Suppose $\psi(\phi) \lhd^* \sigma$ for some ψ. Then $\psi(\phi) \lhd^* \phi$, which is a contradiction to the Reflexive Condition (if ψ is not the identity) or to the assumption that $\sigma \mathrel{<\!\!\lhd} \phi$ (if ψ is the identity). ∎

An abstract domain will be defined for each range type or, equivalently (because of the Simple Range Condition), each constructor. The abstract domain for a range type ϕ depends on $\mathcal{N}(\phi)$. The following lemma provides the guarantee that the abstract domain for ϕ does not accidentally depend on itself.

Lemma 4.4 Suppose for some $l > 1$ there are $C_1, \dots, C_l \in \Sigma_\tau$, tuples of types $\bar{\tau}_1, \dots, \bar{\tau}_l$ and disjoint tuples of distinct parameters $\bar{u}_1, \dots \bar{u}_l$ such that $C_k(\bar{\tau}_k) \mathrel{<\!\!\lhd} C_{k+1}(\bar{u}_{k+1})$ for all $k \in \{1, \dots, l-1\}$. Then $C_1 \neq C_l$.
Proof: $C_k(\bar{\tau}_k) \lhd^+ C_{k+1}(\bar{u}_{k+1})$ for all $k \in \{1, \dots, l-1\}$ by Def. 4.3. By straightforward induction it follows that for all $k, k' \in \{1, \dots, l\}$, where $k < k'$, there is a ψ such that $C_k(\psi(\bar{\tau}_k)) \lhd^+ C_{k'}(\bar{u}_{k'})$.

Now assume that $C_1 = C_l$. Then there is a ψ' such that $C_l(\psi'(\bar{\tau}_1)) \lhd^+ C_{l-1}(\bar{\tau}_{l-1})$ which by Lemma 4.3 is a contradiction to $C_{l-1}(\bar{\tau}_{l-1}) \mathrel{<\!\!\lhd} C_l(\bar{u}_l)$. Hence $C_1 \neq C_l$. ∎

4.2 Traversing concrete terms

In Section 2 we demonstrated the importance of characterising *termination* of a term and instantiation of *subterms* of a term. With the terminology just introduced this can be made more precise. Consider a term t of type ϕ, where ϕ is simple. We are interested in the subterms of t of type σ where either $\sigma \mathrel{<\!\!\lhd} \phi$ or $\sigma \trianglelefteq \phi$. Termination of a term t of type ϕ means that no instance of t has a variable subterm of type ϕ.

Definition 4.4 [recursive subterm] Let ϕ be a simple type and t be a term whose type is an instance of ϕ. We define a sequence $R_0, R_1 \dots$ as follows:

$$\begin{aligned} R_0 &= \{\langle t, \phi \rangle\} \\ R_{k+1} &= \{\langle r_i, \psi(\tau_i) \rangle \mid \langle f_{\langle \tau_1 \dots \tau_n, \tau \rangle}(r_1, \dots, r_n), \psi(\tau) \rangle \in R_k \text{ and } \psi(\tau_i) \trianglelefteq \phi \text{ for some } \psi\}. \end{aligned}$$

If $\langle r, \_\rangle \in R_k$ for some $k \geq 0$ we say that r is a **recursive subterm of** t.

We define two kinds of functions: one to decide termination and the other to extract a set of subterms of a given type. These functions traverse a term following the recursive subterms. We must make sure that, at each step in this traversal, it is well-defined what the recursive subterms *are*. In the above sequence $R_0, R_1 \ldots$, with every *recursive subterm* of t, we need to record the corresponding *recursive type* of ϕ, as well as the initial type ϕ. The functions we define are therefore also parametrised by such a recursive type, and it is always written, as in the above definition, as a declared type with a type substitution applied.

Definition 4.5 [termination function] Let ϕ be a simple type and $f_{\langle \tau_1 \ldots \tau_n, \tau\rangle} \in \Sigma_f$ such that $\psi(\tau) \trianglelefteq \phi$ for some ψ. For a term $t = f(t_1, \ldots, t_n)$ the **termination function** is defined as

$$\mathcal{Z}^{\phi\,\psi(\tau)}(t) = \bigwedge_{\psi(\tau_i) \trianglelefteq \phi} \mathcal{Z}^{\phi\,\psi(\tau_i)}(t_i).$$

If t is a variable (of any type) $\mathcal{Z}^{\phi\,\psi(\tau)}(t) = false$. t is **terminated** if $t = f_{\langle \tau_1 \ldots \tau_n, \tau\rangle}(t_1, \ldots, t_n)$ and $\mathcal{Z}^{\tau\,\tau}(t) = true$. t is **open** if t is not terminated. For a set S of terms $\mathcal{Z}^{\phi\,\psi(\tau)}(S) = \bigwedge_{t \in S} \mathcal{Z}^{\phi\,\psi(\tau)}(t)$.

It is easy to see that a term is open iff it has a recursive subterm which is a variable.

Example 4.4 Consider the declarations of Example 4.3. The term `Tuesday` is terminated. For type `Day` termination is a trivial property because a term of type `Day` can only be open if it is a variable. Variables are always open. The term `EvenCons(Tuesday,OddCons(x,Nil))` is terminated, `EvenCons(Tuesday,OddCons(x,y))` is open.

Given a term t of type ϕ, an extractor for σ is a function which extracts subterms of t which have type σ, where σ is a non-recursive subterm type of ϕ.

Definition 4.6 [extractor for σ] Let ϕ be a simple type, and $f_{\langle \tau_1 \ldots \tau_n, \tau\rangle} \in \Sigma_f$ such that $\psi(\tau) \trianglelefteq \phi$ for some ψ, and $\sigma \in \mathcal{N}(\phi)$. For a term $t = f(t_1, \ldots, t_n)$ the **extractor for** σ is defined as

$$\mathcal{E}^{\sigma\,\phi\,\psi(\tau)}(f(t_1, \ldots, t_n)) = \{t_i \mid \psi(\tau_i) = \sigma\} \;\cup \bigcup_{\psi(\tau_i) \trianglelefteq \phi} \mathcal{E}^{\sigma\,\phi\,\psi(\tau_i)}(t_i).$$

If t is a variable $\mathcal{E}^{\sigma\,\phi\,\psi(\tau)}(t) = \{t\}$. For a set S of terms $\mathcal{E}^{\sigma\,\phi\,\psi(\tau)}(S) = \bigcup_{t \in S} \mathcal{E}^{\sigma\,\phi\,\psi(\tau)}(t)$.

For common examples this formalism turns out to be much simpler than these definitions suggest. We found that writing the types by which these functions are parametrised as superscripts rather than as arguments, it is easier to concentrate on the *terms* on which these functions are defined, and regard the types as a technical detail. We have kept the notation as consistent as possible so that the roles of types σ, ϕ, τ and type substitution ψ are fixed from now on.

Example 4.5 Consider again the `Nests` module of Section 2, and the computation of $\mathcal{E}^{\texttt{v}\,\texttt{Nest(v)}\,\texttt{Nest(v)}}(\texttt{N}([\texttt{E}(7)]))$. In the recursive calls the first and second superscripts stay the same, while the third superscript changes. We go through this computation in detail. The right hand side links this general notation with the example.

$\mathcal{E}^{\texttt{v}\,\texttt{Nest(v)}\,\texttt{Nest(v)}}(\texttt{N}([\texttt{E}(7)])) =$	$\tau = \texttt{Nest(u)},\ \psi = \{\texttt{u/v}\}$
$\mathcal{E}^{\texttt{v}\,\texttt{Nest(v)}\,\texttt{List(Nest(v))}}([\texttt{E}(7)]) =$	$\tau = \texttt{List(u)},\ \psi = \{\texttt{u/Nest(v)}\}$
$\mathcal{E}^{\texttt{v}\,\texttt{Nest(v)}\,\texttt{Nest(v)}}(\texttt{E}(7)) \cup$	$\tau = \texttt{Nest(u)},\ \psi = \{\texttt{u/v}\}$
$\mathcal{E}^{\texttt{v}\,\texttt{Nest(v)}\,\texttt{List(Nest(v))}}(\texttt{Nil}) =$	$\tau = \texttt{List(u)},\ \psi = \{\texttt{u/Nest(v)}\}$
$\{7\} \cup \emptyset = \{7\}.$	

Consider simple types ϕ, τ such that $\psi(\tau) \trianglelefteq \phi$ for some ψ. The following key lemma relates $\mathcal{N}(\phi)$ with $\mathcal{N}(\tau)$. Several subsequent statements will be relatively simple consequences.

Lemma 4.5 Let ϕ and τ be simple types such that $\psi(\tau) \trianglelefteq \phi$ for some ψ, and t be a term whose type is an instance of $\psi(\tau)$. Let $\mathcal{N}(\tau) = \langle \sigma_1, \ldots, \sigma_m \rangle$ and $\sigma \in \mathcal{N}(\phi)$. Then

$$\mathcal{Z}^{\phi\, \psi(\tau)}(t) = \mathcal{Z}^{\tau\, \tau}(t) \wedge \bigwedge_{\psi(\sigma_j) \trianglelefteq \phi} \mathcal{Z}^{\phi\, \psi(\sigma_j)}(\mathcal{E}^{\sigma_j\, \tau\, \tau}(t)) \tag{1}$$

$$\mathcal{E}^{\sigma\, \phi\, \psi(\tau)}(t) = \bigcup_{\psi(\sigma_j) = \sigma} \mathcal{E}^{\sigma_j\, \tau\, \tau}(t) \cup \bigcup_{\psi(\sigma_j) \trianglelefteq \phi} \mathcal{E}^{\sigma\, \phi\, \psi(\sigma_j)}(\mathcal{E}^{\sigma_j\, \tau\, \tau}(t)). \tag{2}$$

Proof: Note that ϕ, τ, σ and ψ occur in the statement and thus are *fixed* throughout the proof. We use f as an abbreviation for $f_{\langle \tau'_1 \ldots \tau'_n, \tau' \rangle}$, and $\bar{r}$ to denote $(r_1, \ldots, r_n)$. We write '_' for a type if it is irrelevant.

The proof is by showing a step-by-step correspondence between the left and right hand side computations. We first explain this informally for (2). For the left hand side, we maintain a sequence of sets R_k of tuples $\langle r, \psi'(\tau') \rangle$, where $\psi'(\tau') \trianglelefteq \phi$, and r is a subterm of t that still must be considered after k steps in the computation of $\mathcal{E}^{\sigma\, \phi\, \psi(\tau)}(t)$. Furthermore, we maintain a sequence of sets S_k, where S_k contains the terms that are extracted in the k-th step. So $\mathcal{E}^{\sigma\, \phi\, \psi(\tau)}(t) = \bigcup_{k \geq 0} S_k$.

For the right hand side, we do something similar but "on two levels". We maintain a sequence of sets A_k of tuples $\langle r, \psi'(\tau') \rangle$, where $\psi'(\tau') \trianglelefteq \tau$ (note: τ, not ϕ!), and r is a subterm of t that still must be considered after k steps in the computations of $\mathcal{E}^{\sigma_j\, \tau\, \tau}(t)$, for all $\sigma_j \in \mathcal{N}(\tau)$ with $\psi(\sigma_j) = \sigma$ or $\psi(\sigma_j) \trianglelefteq \phi$. We maintain just *one* such sequence for all σ_j because this sequence does not depend on σ_j. For all σ_j the subterms of t that are visited in the computation of $\mathcal{E}^{\sigma_j\, \tau\, \tau}(t)$ are the same. If in step k this computation finally arrives at a subterm s which is to be extracted, we must distinguish the cases $\psi(\sigma_j) = \sigma$ and $\psi(\sigma_j) \trianglelefteq \phi$. In the first case, s is inserted into a set $B_k^{\sigma_j}$, which means $s \in \bigcup_{\psi(\sigma_j) = \sigma} \mathcal{E}^{\sigma_j\, \tau\, \tau}(t)$. In the second case, $\mathcal{E}^{\sigma\, \phi\, \psi(\sigma_j)}(s)$ is computed. We insert the tuple $\langle s, \psi(\sigma_j) \rangle$ into a set $C_k^{\sigma_j}$, which has a role similar to R_k and A_k. Finally, we maintain a sequence of sets $D_k^{\sigma_j}$, which has a role similar to S_k and B_k. So $\mathcal{E}^{\sigma\, \phi\, \psi(\sigma_j)}(\mathcal{E}^{\sigma_j\, \tau\, \tau}(t)) = \bigcup_{k \geq 0} D_k^{\sigma_j}$. Formally this is

$$\begin{aligned}
R_0 &= \{\langle t, \psi(\tau) \rangle\} \\
R_k &= \{\langle r_i, \psi'(\tau'_i) \rangle \mid \langle f(\bar{r}), \psi'(\tau') \rangle \in R_{k-1} \text{ and } \psi'(\tau'_i) \trianglelefteq \phi\} \\
S_k &= \{x \mid \langle x, _ \rangle \in R_k\} \ \cup \ \{r_i \mid \langle f(\bar{r}), \psi'(\tau') \rangle \in R_{k-1} \text{ and } \psi'(\tau'_i) = \sigma\} \\
A_0 &= \{\langle t, \tau \rangle\} \\
A_k &= \{\langle r_i, \psi'(\tau'_i) \rangle \mid \langle f(\bar{r}), \psi'(\tau') \rangle \in A_{k-1} \text{ and } \psi'(\tau'_i) \trianglelefteq \tau\}.
\end{aligned}$$

Furthermore for all $\sigma_j \in \mathcal{N}(\tau)$ with $\psi(\sigma_j) = \sigma$ we define

$$B_k^{\sigma_j} = \{x \mid \langle x, \_ \rangle \in A_k\} \ \cup \ \{r_i \mid \langle f(\bar{r}), \psi'(\tau') \rangle \in A_{k-1} \text{ and } \psi'(\tau'_i) = \sigma_j\},$$

and for all $\sigma_j \in \mathcal{N}(\tau)$ with $\psi(\sigma_j) \trianglelefteq \phi$ we define

$$\begin{aligned}
C_k^{\sigma_j} &= \{\langle x, \psi(\psi'(\tau')) \rangle \mid \langle x, \psi'(\tau') \rangle \in A_k\} \cup \\
&\quad\ \{\langle r_i, \psi(\sigma_j) \rangle \mid \langle f(\bar{r}), \psi'(\tau') \rangle \in A_{k-1} \text{ and } \psi'(\tau'_i) = \sigma_j\} \cup \\
&\quad\ \{\langle r_i, \psi'(\tau'_i) \rangle \mid \langle f(\bar{r}), \psi'(\tau') \rangle \in C_{k-1}^{\sigma_j} \text{ and } \psi'(\tau'_i) \trianglelefteq \phi\} \\
D_k^{\sigma_j} &= \{x \mid \langle x, _ \rangle \in C_k^{\sigma_j}\} \cup \{r_i \mid \langle f(\bar{r}), \psi'(\tau') \rangle \in C_{k-1}^{\sigma_j} \text{ and } \psi'(\tau'_i) = \sigma\}.
\end{aligned}$$

Recall that x always denotes a variable. Inspecting Definitions 4.5 and 4.6 it can be stated precisely in what sense these sequences trace the computations of (1) and (2):

- $\mathcal{Z}^{\phi\,\psi(\tau)}(t) = false$ iff $\langle x, \_\rangle \in R_k$ for some $k \geq 0$ and some variable x,
- $\mathcal{Z}^{\tau\,\tau}(t) = false$ iff $\langle x, \_\rangle \in A_k$ for some $k \geq 0$ and some variable x,
- for $\sigma_j \in \mathcal{N}(\tau)$ with $\psi(\sigma_j) \trianglelefteq \phi$, $\mathcal{Z}^{\phi\,\psi(\sigma_j)}(\mathcal{E}^{\sigma_j\,\tau\,\tau}(t)) = false$ iff for some $k \geq 0$ and some variable x, either $\langle x, \_\rangle \in A_k$ or $\langle x, \_\rangle \in C_k^{\sigma_j}$,
- $\mathcal{E}^{\sigma\,\phi\,\psi(\tau)}(t) = \bigcup_{k\geq 0} S_k$,
- $\mathcal{E}^{\sigma_j\,\tau\,\tau}(t) = \bigcup_{k\geq 0} B_k^{\sigma_j}$ for all $\sigma_j \in \mathcal{N}(\tau)$ with $\psi(\sigma_j) = \sigma$,
- $\mathcal{E}^{\sigma\,\phi\,\psi(\sigma_j)}(\mathcal{E}^{\sigma_j\,\tau\,\tau}(t)) = \bigcup_{k\geq 0} D_k^{\sigma_j}$ for all $\sigma_j \in \mathcal{N}(\tau)$ with $\psi(\sigma_j) \trianglelefteq \phi$.

Part 1: We show by induction on k that $\langle r_i, \psi(\psi'(\tau_i))\rangle$[5] $\in R_k$ iff $\langle r_i, \psi'(\tau_i)\rangle \in A_k$ or $\langle r_i, \psi(\psi'(\tau_i'))\rangle \in C_k^{\sigma_j}$ for some $\sigma_j \in \mathcal{N}(\tau)$ with $\psi(\sigma_j) \trianglelefteq \phi$. For $k = 0$ this is true.

"$\Rightarrow$": Assume $\langle r_i, \psi(\psi'(\tau_i'))\rangle \in R_{k+1}$. By definition $\psi(\psi'(\tau_i')) \trianglelefteq \phi$ and there is $\langle f(\bar{r}), \psi(\psi'(\tau'))\rangle \in R_k$. By hypothesis either $\langle f(\bar{r}), \psi'(\tau')\rangle \in A_k$, or $\langle f(\bar{r}), \psi(\psi'(\tau'))\rangle \in C_k^{\sigma_j}$ for some $\sigma_j \in \mathcal{N}(\tau)$ with $\psi(\sigma_j) \trianglelefteq \phi$.

Consider first $\langle f(\bar{r}), \psi'(\tau')\rangle \in A_k$. Since $\psi'(\tau') \trianglelefteq \tau$ (this is an invariant of the sequence A_k), either $\psi'(\tau_i') \trianglelefteq \tau$ or $\psi'(\tau_i') \lll \tau$. If $\psi'(\tau_i') \trianglelefteq \tau$ then $\langle r_i, \psi'(\tau_i')\rangle \in A_{k+1}$. If $\psi'(\tau_i') \lll \tau$ then $\psi'(\tau_i') = \sigma_j$ for some $\sigma_j \in \mathcal{N}(\tau)$. Furthermore $\psi(\sigma_j) \trianglelefteq \phi$, since $\psi(\psi'(\tau_i')) \trianglelefteq \phi$. This implies $\langle r_i, \psi(\psi'(\tau_i'))\rangle \in C_{k+1}^{\sigma_j}$.

Consider secondly $\langle f(\bar{r}), \psi(\psi'(\tau'))\rangle \in C_k^{\sigma_j}$ for some $\sigma_j \in \mathcal{N}(\tau)$ with $\psi(\sigma_j) \trianglelefteq \phi$. Then $\langle r_i, \psi(\psi'(\tau_i'))\rangle \in C_{k+1}^{\sigma_j}$, since $\psi(\psi'(\tau_i')) \trianglelefteq \phi$.

"$\Leftarrow$": Assume $\langle r_i, \psi'(\tau_i')\rangle \in A_{k+1}$. By definition there is $\langle f(\bar{r}), \psi'(\tau')\rangle \in A_k$ and $\psi'(\tau_i') \trianglelefteq \tau$. By induction hypothesis $\langle f(\bar{r}), \psi(\psi'(\tau'))\rangle \in R_k$. Since $\psi'(\tau_i') \trianglelefteq \tau$ and $\psi(\tau) \trianglelefteq \phi$ by Lemma 4.2 it follows that $\langle r_i, \psi(\psi'(\tau_i'))\rangle \in R_{k+1}$.

Now assume $\langle r_i, \psi(\psi'(\tau_i'))\rangle \in C_{k+1}^{\sigma_j}$ for some $\sigma_j \in \mathcal{N}(\tau)$ with $\psi(\sigma_j) \trianglelefteq \phi$. By definition either r_i is a variable and $\langle r_i, \psi'(\tau_i')\rangle \in A_{k+1}$, or there is $\langle f(\bar{r}), \psi'(\tau')\rangle \in A_k$ and $\psi'(\tau_i') = \sigma_j$, or there is $\langle f(\bar{r}), \psi(\psi'(\tau'))\rangle \in C_k^{\sigma_j}$ and $\psi(\psi'(\tau_i')) \trianglelefteq \phi$. The first case has already been shown. In the second case, by induction hypothesis $\langle f(\bar{r}), \psi(\psi'(\tau'))\rangle \in R_k$, and putting together that $\psi'(\tau_i') = \sigma_j$ and $\psi(\sigma_j) \trianglelefteq \phi$, it follows that $\langle r_i, \psi(\psi'(\tau_i'))\rangle \in R_{k+1}$. In the third case, by induction hypothesis $\langle f(\bar{r}), \psi(\psi'(\tau'))\rangle \in R_k$, and since $\psi(\psi'(\tau_i')) \trianglelefteq \phi$, $\langle r_i, \psi(\psi'(\tau_i'))\rangle \in R_{k+1}$.

Part 2: We show (1). $\mathcal{Z}^{\phi\,\psi(\tau)}(t) = false$, iff $\langle x, \_\rangle \in R_k$ for some $k \geq 0$ and some variable x, iff $\langle x, \_\rangle \in A_k$ or$\langle x, _\rangle \in C_k^{\sigma_j}$ for some $\sigma_j \in \mathcal{N}(\tau)$ with $\psi(\sigma_j) \trianglelefteq \phi$, iff $\mathcal{Z}^{\tau\,\tau}(t) \wedge \bigwedge_{\psi(\sigma_j) \trianglelefteq \phi} \mathcal{Z}^{\phi\,\psi(\sigma_j)}(\mathcal{E}^{\sigma_j\,\tau\,\tau}(t)) = false$.

Part 3: We show (2) by showing that for all $k \geq 0$: $S_k = \bigcup\limits_{\psi(\sigma_j)=\sigma} (B_k^{\sigma_j}) \cup \bigcup\limits_{\psi(\sigma_j) \trianglelefteq \phi} (D_k^{\sigma_j})$.

"$\subset$": Assume $r_i \in S_k$. By definition either r_i is a variable x and $\langle x, \_\rangle \in R_k$, or there is $\langle f(\bar{r}), \psi(\psi'(\tau'))\rangle \in R_{k-1}$ and $\psi(\psi'(\tau_i')) = \sigma$. In the first case by Part 1, either $\langle x, \_\rangle \in A_k$, or $\langle x, \_\rangle \in C_k^{\sigma_j}$ for some $\sigma_j \in \mathcal{N}(\tau)$ with $\psi(\sigma_j) \trianglelefteq \phi$. Thus $x \in B_k$ or $x \in D_k^{\sigma_j}$. In the second case by Part 1, either $\langle f(\bar{r}), \psi'(\tau')\rangle \in A_{k-1}$, or $\langle f(\bar{r}), \psi(\psi'(\tau'))\rangle \in C_{k-1}^{\sigma_j}$ for some $\sigma_j \in \mathcal{N}(\tau)$ with $\psi(\sigma_j) \trianglelefteq \phi$.

If $\langle f(\bar{r}), \psi'(\tau')\rangle \in A_{k-1}$, then since $\psi(\psi'(\tau_i')) \lll \phi$ it follows by Lemma 4.2 that $\psi'(\tau_i') \in \mathcal{N}(\tau)$. Thus there exists $\sigma_j \in \mathcal{N}(\tau)$ with $\sigma_j = \psi'(\tau_i')$. Furthermore since $\psi(\sigma_j) = \sigma$, it follows that $r_i \in B_k^{\sigma_j}$. If $\langle f(\bar{r}), \psi(\psi'(\tau'))\rangle \in C_{k-1}^{\sigma_j}$ for some $\sigma_j \in \mathcal{N}(\tau)$ with $\psi(\sigma_j) \trianglelefteq \phi$, then since $\psi(\psi'(\tau_i')) = \sigma$ it follows that $r_i \in D_k^{\sigma_j}$.

"$\supset$": First assume $r_i \in B_k^{\sigma_j}$ for some $\sigma_j \in \mathcal{N}(\tau)$ with $\psi(\sigma_j) = \sigma$. By definition either r_i is a variable x and $\langle x, \_\rangle \in A_k$, or there is $\langle f(\bar{r}), \psi'(\tau')\rangle \in A_{k-1}$ and $\psi'(\tau_i) = \sigma_j$. In the first case by Part 1 $\langle x, \_\rangle \in R_k$ and thus $x \in S_k$. In the second case by Part 1 $\langle f(\bar{r}), \psi(\psi'(\tau'))\rangle \in R_{k-1}$, and putting together that $\psi(\sigma_j) = \sigma$ and $\sigma_j = \psi'(\tau_i)$ it follows that $r_i \in S_k$.

[5] Note that the type in the second position of the tuple is intentionally denoted in this way. Either τ_i' occurs as argument type of some function f and $\langle f(\bar{r}), \psi(\psi'(\tau'))\rangle \in R_{k-1}$, or $\tau_i' = \tau$ and ψ' is empty.

Secondly assume $r_i \in D_k^{\sigma_j}$ for some $\sigma_j \in \mathcal{N}(\tau)$ with $\psi(\sigma_j) \trianglelefteq \phi$. By definition either r_i is a variable x and $\langle x, \_\rangle \in C_k^{\sigma_j}$, or there is $\langle f(\bar r), \psi(\psi'(\tau'))\rangle \in C_{k-1}^{\sigma_j}$ and $\psi(\psi'(\tau'_i)) = \sigma$. In the first case by Part 1 $\langle x, \_\rangle \in R_k$ and thus $x \in S_k$. In the second case by Part 1 $\langle f(\bar r), \psi(\psi'(\tau'))\rangle \in R_{k-1}$, and since $\psi(\psi'(\tau'_i)) = \sigma$, it follows that $r_i \in S_k$. ∎

Example 4.6 Consider the second line of the equation sequence in Example 4.5. Lemma 4.5 says that $\mathcal{E}^{\mathtt{v}\ \mathtt{Nest(v)}\ \mathtt{List(Nest(v))}}(\mathtt{[E(7)]}) = \mathcal{E}^{\mathtt{v}\ \mathtt{Nest(v)}\ \mathtt{Nest(v)}}(\mathcal{E}^{\mathtt{u}\ \mathtt{List(u)}\ \mathtt{List(u)}}(\mathtt{[E(7)]}))$. Indeed $\mathcal{E}^{\mathtt{v}\ \mathtt{Nest(v)}\ \mathtt{Nest(v)}}(\mathcal{E}^{\mathtt{u}\ \mathtt{List(u)}\ \mathtt{List(u)}}(\mathtt{[E(7)]})) = \mathcal{E}^{\mathtt{v}\ \mathtt{Nest(v)}\ \mathtt{Nest(v)}}(\mathtt{E(7)}) = \{7\}$. Here $\phi = \mathtt{Nest(v)}$, $\sigma = \mathtt{v}$, $\tau = \mathtt{List(u)}$, $\mathcal{N}(\tau) = \langle \mathtt{u}\rangle$, and $\psi = \{\mathtt{u/Nest(v)}\}$. Thus Lemma 4.5 relates $\mathcal{N}(\mathtt{Nest(v)})$ with $\mathcal{N}(\mathtt{List(u)})$.

In the next section we will define abstract terms and the usual order $<$ on abstract terms: smaller abstract terms correspond to more instantiated concrete terms. The abstraction of terms relies on extractors. Thus it is important that the property "$\mathcal{E}^{\sigma\ \phi\ \psi(\tau)}(t)$ does not contain variables" is instantiation-closed. This property is a consequence of the following lemma.

Lemma 4.6 Let ϕ and τ be simple types such that $\psi(\tau) \trianglelefteq \phi$ for some ψ, and t be a term whose type is an instance of $\psi(\tau)$, and $\sigma \in \mathcal{N}(\phi)$. If $\mathcal{E}^{\sigma\ \phi\ \psi(\tau)}(t)$ does not contain variables, then $\mathcal{E}^{\sigma\ \phi\ \psi(\tau)}(t)\rho = \mathcal{E}^{\sigma\ \phi\ \psi(\tau)}(t\rho)$ for all substitutions ρ.

Proof: The proof is by structural induction on t. If t is a variable or constant, the result is obvious. Suppose $t = f_{\langle \tau_1 \dots \tau_n, \tau\rangle}(t_1, \dots, t_n)$ where $n > 0$. Then

$$\begin{aligned}
\mathcal{E}^{\sigma\ \phi\ \psi(\tau)}(t)\,\rho &= \{t_i \mid \psi(\tau_i) = \sigma\}\,\rho \;\cup \bigcup_{\psi(\tau_i) \trianglelefteq \phi} \mathcal{E}^{\sigma\ \phi\ \psi(\tau_i)}(t_i)\rho \\
&\overset{hyp.}{=} \{t_i\rho \mid \psi(\tau_i) = \sigma\} \;\cup \bigcup_{\psi(\tau_i) \trianglelefteq \phi} \mathcal{E}^{\sigma\ \phi\ \psi(\tau_i)}(t_i\rho) \;=\; \mathcal{E}^{\sigma\ \phi\ \psi(\tau)}(t\rho).
\end{aligned}$$

∎

5 Abstract Terms and Abstract Programs

In this section we define an abstract domain for each type. Then an abstraction function α for terms is defined, which means no more than formalising a description of terms. This abstraction is extended to atoms and sets of atoms. Thus, for an *interpretation* (a set of concrete atoms), there is a corresponding *abstract interpretation* (a set of abstract atoms). On the other hand there is an abstraction function for *programs*, which also involves abstracting atoms. To avoid confusion between the two functions, we call the latter $\aleph$. The two functions are related in the following sense: The abstraction α of the interpretation of a program is a subset of the interpretation of the abstraction $\aleph$ of the program.

5.1 Abstract domains

Definition 5.1 [abstract domain] Let $\tau = C(\bar u)$ be a simple type, $\mathcal{N}(\tau) = \langle \sigma_1, \dots, \sigma_m\rangle$, and ψ be a type substitution. The **abstract domain $D^{\mathcal{A}}_{\psi(\tau)}$ for $\psi(\tau)$** is defined as

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

For a parameter u define $D^{\mathcal{A}}_u = \{\mathtt{Any},\ \mathtt{Bot}\}$. If $b \in D^{\mathcal{A}}_{\psi(\tau)}$ then b is an **abstract term for $\psi(\tau)$**.

`Ter` and `Open` are called termination flags. A termination flag is not an abstract term, but occurs in an abstract term in the last argument position to represent termination for the corresponding concrete term. Termination flags can be thought of as *truth values* of the property "termination". Thus we can identify `Ter` with *true* and `Open` with *false* and use $\wedge$.

Definition 5.2 [order $<$ on abstract terms] First define $\mathtt{Ter} < \mathtt{Open}$. For abstract terms define $b \leq b'$ if either $b = \mathtt{Bot}$, or $b' = \mathtt{Any}$, or $b = C^{\mathcal{A}}(b_1, \dots, b_m, \mathit{fl})$, $b' = C^{\mathcal{A}}(b'_1, \dots, b'_m, \mathit{fl}')$, and $\mathit{fl} \leq \mathit{fl}'$ and $b_j \leq b'_j$ for all $j \in \{1, \dots, m\}$. For a set S of abstract terms or termination flags, $\sqcup S$ denotes the least upper bound. Note that $\sqcup \emptyset = \mathtt{Bot}$.

The order $<$ should not be associated in any way with the order $\prec$ (see Section 3).

5.2 Abstraction of terms

The abstract term for a variable is $\mathtt{Any}$. For a term $t = f_{\langle \tau_1 \dots \tau_n, \tau \rangle}(t_1, \dots, t_n)$, where $\tau = C(\bar{u})$, the abstract term has the form $C^{\mathcal{A}}(\dots)$. The arguments are abstractions of subterms of t, as well as a termination flag. So $C^{\mathcal{A}}(\dots)$ characterises the instantiation of t in a precise way.

Definition 5.3 [abstraction function α for terms] To save stating the base case explicitly we first lift α to *sets* of terms: $\alpha(S) = \{\alpha(t) \mid t \in S\}$. For truth values define $\alpha(\mathit{true}) = \mathtt{Ter}$ and $\alpha(\mathit{false}) = \mathtt{Open}$. If t is a variable define $\alpha(t) = \mathtt{Any}$. If $t = f_{\langle \tau_1 \dots \tau_n, \tau \rangle}(t_1, \dots, t_n)$, where $\tau = C(\bar{u})$ and $\mathcal{N}(\tau) = \langle \sigma_1, \dots, \sigma_m \rangle$, define

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

For atoms define $\alpha(p_{\langle \tau_1 \dots \tau_n \rangle}(t_1, \dots, t_n)) = p_{\langle \tau_1 \dots \tau_n \rangle}(\alpha(t_1), \dots, \alpha(t_n))$, and for sets of atoms I define $\alpha(I) = \{\alpha(a) \mid a \in I\}$.

The base case is given when τ is only inhabited by constants, in which case $m = 0$. Then $\alpha(t) = \mathtt{Any}$ if $t \in V$, and $\alpha(t) = \mathtt{C}^{\mathcal{A}}(\mathtt{Ter})$ if $t \in \Sigma_f$. This is the abstraction we used for `Integer` in Section 2, except that we now always provide a termination flag even where it is trivial, to have a uniform notation. In program text the superscript $^{\mathcal{A}}$ is denoted by `_a`.

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. E.g. $\alpha(\mathtt{Nil}) = \mathtt{List}^{\mathcal{A}}(\mathtt{Bot}, \mathtt{Ter})$, where `Nil` is defined as usual. `Bot` can be thought of as characterising that all subterms of `Nil` are ground, which is safe.

Example 5.1 We illustrate Def. 5.3 with Example 4.3. For $t = \mathtt{Friday}$ we have $\tau = \mathtt{Day}$, $m = 0$ and $n = 0$. Then $\alpha(\mathtt{Friday}) = \mathtt{Day\_a}(\mathtt{Ter})$. For $t = \mathtt{Nil}$ we have $\tau = \mathtt{EvenList(u)}$, $\mathcal{N}(\mathtt{EvenList(u)}) = \{\mathtt{u}\}$, and $n = 0$. Thus $\alpha(\mathtt{Nil}) = \mathtt{EvenList\_a}(\sqcup(\emptyset), \alpha(\mathcal{Z}^{\tau\,\tau}(\mathtt{Nil}))) = \mathtt{EvenList\_a}(\mathtt{Bot}, \mathtt{Ter})$. Finally

$$\begin{aligned}\alpha(\mathtt{OddCons(Friday, Nil)}) &= \mathtt{OddList\_a}(\sqcup(\{\alpha(\mathtt{Friday})\}), \alpha(\mathcal{Z}^{\tau\,\tau}(\mathtt{OddCons(Friday, Nil)})))\\ &= \mathtt{OddList\_a}(\mathtt{Day\_a}(\mathtt{Ter}), \mathtt{Ter}).\end{aligned}$$

5.3 Traversing abstract terms

Recall the termination functions (Def. 4.5) and extractors (Def. 4.6) defined for concrete terms. We define corresponding functions for *abstract* terms.

Definition 5.4 [Abstract termination function; abstract extractor for σ] Let ϕ and $\tau = C(\bar{u})$ be simple types such that $\psi(\tau) \trianglelefteq \phi$ for some ψ, and $\mathcal{N}(\tau) = \langle \sigma_1, \dots, \sigma_m \rangle$. Let $b = C^{\mathcal{A}}(b_1, \dots, b_m, c)$.

1. Abstract termination function.

$$\mathcal{AZ}^{\phi\, \psi(\tau)}(b) = \bigwedge_{\psi(\sigma_j) \trianglelefteq \phi} \mathcal{AZ}^{\phi\, \psi(\sigma_j)}(b_j) \wedge c$$

Furthermore $\mathcal{AZ}^{\phi\,\psi(\tau)}(\texttt{Any}) = \texttt{Open}$ and $\mathcal{AZ}^{\phi\,\psi(\tau)}(\texttt{Bot}) = \texttt{Ter}$.
2. Abstract extractor for σ. Let $\sigma \in \mathcal{N}(\phi)$.

$$\mathcal{AE}^{\sigma\,\phi\,\psi(\tau)}(b) = \{b_j \mid \psi(\sigma_j) = \sigma\} \cup \bigcup_{\psi(\sigma_j) \trianglelefteq \phi} \mathcal{AE}^{\sigma\,\phi\,\psi(\sigma_j)}(b_j)$$

Furthermore $\mathcal{AE}^{\sigma\,\phi\,\psi(\tau)}(\texttt{Any}) = \{\texttt{Any}\}$ and $\mathcal{AE}^{\sigma\,\phi\,\psi(\tau)}(\texttt{Bot}) = \emptyset$.

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

Lemma 5.1 Let ϕ and $\tau = C(\bar{u})$ be simple types such that $\psi(\tau) \trianglelefteq \phi$ for some ψ, and $\sigma \in \mathcal{N}(\phi)$. Let t be a term whose type is an instance of $\psi(\tau)$. Then

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

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

Proof: The proof is by induction on the structure of t. First assume t is a variable x or a constant c.

$$\begin{aligned} \alpha(\mathcal{Z}^{\phi\,\psi(\tau)}(x)) &= \alpha(false) = \texttt{Open} = \mathcal{AZ}^{\phi\,\psi(\tau)}(\texttt{Any}) = \mathcal{AZ}^{\phi\,\psi(\tau)}(\alpha(x)). \\ \sqcup\,\alpha(\mathcal{E}^{\sigma\,\phi\,\psi(\tau)}(x)) &= \sqcup\,\{\alpha(x)\} = \texttt{Any} = \sqcup\,\mathcal{AE}^{\sigma\,\phi\,\psi(\tau)}(\texttt{Any}) = \sqcup\,\mathcal{AE}^{\sigma\,\phi\,\psi(\tau)}(\alpha(x)). \end{aligned}$$

$$\begin{aligned} \alpha(\mathcal{Z}^{\phi\,\psi(\tau)}(c)) &= \alpha(true) = \texttt{Ter} = \mathcal{AZ}^{\phi\,\psi(\tau)}(C^{\mathcal{A}}(\texttt{Bot},\ldots,\texttt{Bot},\texttt{Ter})) = \mathcal{AZ}^{\phi\,\psi(\tau)}(\alpha(c)). \\ \sqcup\,\alpha(\mathcal{E}^{\sigma\,\phi\,\psi(\tau)}(c)) &= \sqcup\,\emptyset = \texttt{Bot} = \sqcup\,\mathcal{AE}^{\sigma\,\phi\,\psi(\tau)}(C^{\mathcal{A}}(\texttt{Bot},\ldots,\texttt{Bot},\texttt{Ter})) = \sqcup\,\mathcal{AE}^{\sigma\,\phi\,\psi(\tau)}(\alpha(c)). \end{aligned}$$

Now assume t is a compound term. In the following sequences of equations, * marks steps which use straightforward manipulations such as rearranging least upper bounds or applications of α to sets. We show (1) working from right to left. By Def. 5.3

$$\begin{aligned} \mathcal{AZ}^{\phi\,\psi(\tau)}(\alpha(t)) &= \mathcal{AZ}^{\phi\,\psi(\tau)}(C^{\mathcal{A}}(\sqcup\alpha(\mathcal{E}^{\sigma_1\,\tau\,\tau}(t)),\ldots,\sqcup\alpha(\mathcal{E}^{\sigma_m\,\tau\,\tau}(t)),\alpha(\mathcal{Z}^{\tau\,\tau}(t)))) \\ \text{(Def. 5.4)} \quad &= \alpha(\mathcal{Z}^{\tau\,\tau}(t)) \wedge \bigwedge_{\psi(\sigma_j) \trianglelefteq \phi} \mathcal{AZ}^{\phi\,\psi(\sigma_j)}(\sqcup\alpha(\mathcal{E}^{\sigma_j\,\tau\,\tau}(t))) \\ (*) \quad &= \alpha(\mathcal{Z}^{\tau\,\tau}(t)) \wedge \bigwedge_{\psi(\sigma_j) \trianglelefteq \phi} \mathcal{AZ}^{\phi\,\psi(\sigma_j)}(\alpha(\mathcal{E}^{\sigma_j\,\tau\,\tau}(t))) \\ \text{(hypothesis)} \quad &= \alpha(\mathcal{Z}^{\tau\,\tau}(t)) \wedge \bigwedge_{\psi(\sigma_j) \trianglelefteq \phi} \alpha(\mathcal{Z}^{\phi\,\psi(\sigma_j)}(\mathcal{E}^{\sigma_j\,\tau\,\tau}(t))) \\ \text{(Lemma 4.5)} \quad &= \alpha(\mathcal{Z}^{\phi\,\psi(\tau)}(t)). \end{aligned}$$

We show (2), also working from right to left. By Def. 5.3

$$\begin{aligned} \sqcup\,\mathcal{AE}^{\sigma\,\phi\,\psi(\tau)}(\alpha(t)) &= \sqcup\,\mathcal{AE}^{\sigma\,\phi\,\psi(\tau)}(C^{\mathcal{A}}(\sqcup\alpha(\mathcal{E}^{\sigma_1\,\tau\,\tau}(t)),\ldots,\sqcup\alpha(\mathcal{E}^{\sigma_m\,\tau\,\tau}(t)),\alpha(\mathcal{Z}^{\tau\,\tau}(t)))) \\ \text{(Def. 5.4)} \quad &= \sqcup\,(\{\sqcup\alpha(\mathcal{E}^{\sigma_j\,\tau\,\tau}(t)) \mid \psi(\sigma_j) = \sigma\} \cup \bigcup_{\psi(\sigma_j) \trianglelefteq \phi} \mathcal{AE}^{\sigma\,\phi\,\psi(\sigma_j)}(\sqcup\,\alpha(\mathcal{E}^{\sigma_j\,\tau\,\tau}(t)))) \\ (*) \quad &= \sqcup\,(\bigcup_{\psi(\sigma_j)=\sigma} \alpha(\mathcal{E}^{\sigma_j\,\tau\,\tau}(t)) \cup \bigcup_{\psi(\sigma_j) \trianglelefteq \phi} \{\sqcup\,\mathcal{AE}^{\sigma\,\phi\,\psi(\sigma_j)}(\alpha(\mathcal{E}^{\sigma_j\,\tau\,\tau}(t)))\}) \\ \text{(hypothesis)} \quad &= \sqcup\,(\bigcup_{\psi(\sigma_j)=\sigma} \alpha(\mathcal{E}^{\sigma_j\,\tau\,\tau}(t)) \cup \bigcup_{\psi(\sigma_j) \trianglelefteq \phi} \{\sqcup\,\alpha(\mathcal{E}^{\sigma\,\phi\,\psi(\sigma_j)}(\mathcal{E}^{\sigma_j\,\tau\,\tau}(t)))\}) \\ \text{(* \& Lemma 4.5)} \quad &= \sqcup\,\alpha(\mathcal{E}^{\sigma\,\phi\,\psi(\tau)}(t)). \end{aligned}$$

∎

5.4 Abstraction of programs

The essential part of abstracting a program is abstracting equations[6]. All unifications in a concrete (normal form) program are made explicit by equations, and the *abstracted* equations define corresponding unifications between abstract terms. First we define 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 [abstract dependency f_{dep}] Let $f_{\langle \tau_1 \ldots \tau_n, \tau \rangle} \in \Sigma_f$ where $\tau = C(\bar{u})$ and $\mathcal{N}(\tau) = \langle \sigma_1, \ldots, \sigma_m \rangle$. $f_{\text{dep}}(C^{\mathcal{A}}(a_1, \ldots, a_m, c), b_1, \ldots, b_n)$ holds if

$$a_j \;=\; \sqcup(\{b_i \mid \tau_i = \sigma_j\} \cup \bigcup_{\tau_i \unlhd \tau} \mathcal{A}\mathcal{E}^{\sigma_j\,\tau\,\tau_i}(b_i)) \quad \text{for all } j \in \{1, \ldots, m\} \tag{1}$$

$$c \;=\; \bigwedge_{\tau_i \unlhd \tau} \mathcal{A}\mathcal{Z}^{\tau\,\tau_i}(b_i). \tag{2}$$

The following theorem states the relationship between concrete and abstract equations.

Theorem 5.2 If $t = f_{\langle \tau_1 \ldots \tau_n, \tau \rangle}(t_1, \ldots, t_n)$ then $f_{\text{dep}}(\alpha(t), \alpha(t_1), \ldots, \alpha(t_n))$ holds.
Proof: By Def. 5.3, $\alpha(t) = C^{\mathcal{A}}(\sqcup\alpha(\mathcal{E}^{\sigma_1 \tau\tau}(t)), \ldots, \sqcup\alpha(\mathcal{E}^{\sigma_m \tau\tau}(t)), \alpha(\mathcal{Z}^{\tau\,\tau}(t)))$. We have to verify (1) and (2) of Def. 5.5. By Def. 4.6, for all $j \in \{1, \ldots, m\}$

$$\begin{array}{lrl}
 & \sqcup\,\alpha(\mathcal{E}^{\sigma_j\,\tau\tau}(t)) = & \sqcup\,\alpha(\{t_i \mid \tau_i = \sigma_j\} \;\cup \bigcup_{\tau_i \unlhd \tau} \mathcal{E}^{\sigma_j\,\tau\,\tau_i}(t_i)) \\
\text{(moving } \alpha \text{ inwards)} & = & \sqcup(\{\alpha(t_i) \mid \tau_i = \sigma_j\} \cup \bigcup_{\tau_i \unlhd \tau} \alpha(\mathcal{E}^{\sigma_j\,\tau\,\tau_i}(t_i))) \\
\text{(Lemma 5.1)} & = & \sqcup(\{\alpha(t_i) \mid \tau_i = \sigma_j\} \cup \bigcup_{\tau_i \unlhd \tau} \mathcal{A}\mathcal{E}^{\sigma_j\,\tau\,\tau_i}(\alpha(t_i))),
\end{array}$$

so (1) holds. We now show (2):

$$\alpha(\mathcal{Z}^{\tau\,\tau}(t)) = \alpha(\bigwedge_{\tau_i \unlhd \tau} \mathcal{Z}^{\tau\,\tau_i}(t_i)) = \bigwedge_{\tau_i \unlhd \tau} \alpha(\mathcal{Z}^{\tau\,\tau_i}(t_i)) \stackrel{\text{Lemma 5.1}}{=} \bigwedge_{\tau_i \unlhd \tau} \mathcal{A}\mathcal{Z}^{\tau\,\tau_i}(\alpha(t_i)).$$

■

Definition 5.6 [abstraction of a program] For a normalised equation e (see Section 2) define

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

For an atom a define $\aleph(a) = a$. For a clause $K = h \leftarrow g_1 \wedge \ldots \wedge g_l$ define $\aleph(K) = \aleph(h) \leftarrow \aleph(g_1) \wedge \ldots \wedge \aleph(g_l)$. For a set S of clauses define $\aleph(S) = \{\aleph(K) \mid K \in S\}$. For a program $P = \langle L, S \rangle$ define $\aleph(P) = \aleph(S) \cup \{f_{\text{dep}}(a, a_1, \ldots, a_n) \mid f_{\text{dep}}(a, a_1, \ldots, a_n) \text{ holds}\}$.

5.5 Correspondence between concrete and abstract programs

We assume a fixed language L and a program $P = \langle L, S \rangle$. For a substitution θ define $\theta^\alpha = \{x/\alpha(x\theta) \mid x \in V\}$. G and H denote goals (= sequences of atoms). We use an operational semantics with left-to-right computation rule, in which a *program state* is a tuple $\langle G, \theta \rangle$ where G is a goal and θ is a substitution. An initial state is a state where θ is empty.

[6] "Built-ins", i.e. predicates provided by the system, have been considered in the implementation (see Section 6), but we ignore them here.

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

$$\langle h_1 : \ldots : h_l, \theta\rangle \xrightarrow{P} \langle h_2 : \ldots : h_l, \theta\theta'\rangle \quad \text{if} \quad h_1 \text{ is '}x = t\text{' and } x\theta\theta' = t\theta\theta' \tag{1}$$

$$\langle h_1 : \ldots : h_l, \theta\rangle \xrightarrow{P} \langle G : h_2 : \ldots : h_l, \theta\theta'\rangle \quad \text{if} \quad h \leftarrow G \text{ is a renamed variant from } S \text{ and } h\theta\theta' = h_1\theta\theta' \tag{2}$$

$\xrightarrow{P}{}^j$ for $j \geq 0$ and $\xrightarrow{P}{}^*$ are defined in the usual way.

This is standard [5, 11] except that we do not require θ' to be the most general unifier. However our notion of "reduces" is considered in [10].

Theorem 5.3 Let H be a sequence of normalised atoms, θ be a substitution and $j \geq 0$. If $\langle H, \emptyset\rangle \xrightarrow{P}{}^j \langle H', \theta\rangle$ then $\langle \aleph(H), \emptyset\rangle \xrightarrow{\aleph(P)}{}^j \langle \aleph(H'), \theta^\alpha\rangle$.
Proof: By Def. 5.7 $\langle H, \emptyset\rangle \xrightarrow{P}{}^j \langle H', \theta\rangle$ iff $\langle H, \theta\rangle \xrightarrow{P}{}^j \langle H', \theta\rangle$, and likewise for $\aleph(P)$. Therefore we show that $\langle H, \theta\rangle \xrightarrow{P}{}^j \langle H', \theta\rangle$ implies $\langle \aleph(H), \theta^\alpha\rangle \xrightarrow{\aleph(P)}{}^j \langle \aleph(H'), \theta^\alpha\rangle$.

The proof is by induction on j. The base case $j = 0$ is trivial. We must show that $\langle H, \theta\rangle \xrightarrow{P} \langle H', \theta\rangle$ implies $\langle \aleph(H), \theta^\alpha\rangle \xrightarrow{\aleph(P)} \langle \aleph(H'), \theta^\alpha\rangle$. We distinguish Rules (1) and (2).
(1): Here t could be a variable y or a term $f(x_1, \ldots, x_n)$. If $t = y$, then $\aleph(h_1) = h_1$. Since $x\theta = y\theta$ we have $\{x/\alpha(x\theta), y/\alpha(x\theta)\} \subseteq \theta^\alpha$ and hence $x\theta^\alpha = y\theta^\alpha$. If $t = f(x_1, \ldots, x_n)$, then $\aleph(h_1) = f_{\mathrm{dep}}(x, x_1, \ldots, x_n)$. Since $x\theta = f(x_1\theta, \ldots, x_n\theta)$ we have $\{x/\alpha(f(x_1\theta, \ldots, x_n\theta)), x_1/\alpha(x_1\theta), \ldots, x_n/\alpha(x_n\theta)\} \subseteq \theta^\alpha$ and hence $f_{\mathrm{dep}}(x, x_1, \ldots, x_n)\,\theta^\alpha \in \aleph(P)$ by Theorem 5.2. In this case the reduction step in $\aleph(P)$ is by Rule (2).
(2): Clearly $\aleph(h \leftarrow G) \in \aleph(P)$. Furthermore $\aleph(h)$ has the form $Q(\bar{x})$ and $\aleph(h_1)$ has the form $Q(\bar{y})$. Since $\bar{x}\theta = \bar{y}\theta$ we have $Q(\bar{x})\,\theta^\alpha = Q(\bar{y})\,\theta^\alpha$. ■

6 Implementation

The implementation of our mode analysis naturally falls into two stages: On the one hand there is the analysis of the type declarations and the abstraction of the program statements. On the other hand there is the evaluation of the abstract program.

We have implemented the first stage of the analysis in Gödel, using the Gödel meta-programming facilities, which operate on a *ground representation* of the program being analysed. There is a clean separation between the meta and object levels, unlike meta-programming in Prolog where variables in the analysed program are usually represented by variables.

Currently this implementation consists of about 2400 lines of code, of which 650 lines deal with the special treatment of Gödel system modules. The types and predicates provided by the system modules need only be analysed and abstracted once and for all. Furthermore the implementation of the system predicates is hidden and thus not accessible to the meta-programming facilities. Therefore they had to be abstracted by hand according to their specifications.

Gödel meta-programming is rather slow. Even for simple programs, analysing the type declarations and generating an abstract program takes many seconds. However, there is no doubt that the code could be made more efficient, and of course, we could re-implement the analyser in another language. Our examples were simple only in that the programs contained few statements. The type declarations were much more complex than anything one would ever expect in practice! Since abstracting the program statements is relatively trivial, it seems reasonable to expect that generating abstract programs scales well. Furthermore many programs might use the same types and thus the abstract domains would not have to be re-generated.

Program:		append	reverse	flatten	tree_to_list	quicksort
("two-point" domain) {	(1)	1	1	1	1	1
	(2)	1.4	1.4	1.6	1.7	3.4
(our domains) {	(3)	1.4	1.0	1.5	3.3	1.4
	(4)	3.2	2.0	2.8	12	8.7

Table 1: Analysis times

Now consider the evaluation of the abstract program. In order to give an idea of the cost of a mode analysis using our domains, we transformed a few abstract programs into Prolog. We compared the cost of using our domains with the cost using a domain which can only distinguish groundness and non-groundness [4] (which is essentially the *Pos* domain [12]).

Before looking at the figures let us recall what the precision of our domains means. Whenever an analysis using our domains infers that an argument is partially instantiated but not necessarily ground, then a ground/non-ground domain cannot infer anything at all and thus is strictly less precise than our domain. This has two implications. First, if a particular program and initial goal have the property that arguments are either ground or free at most program points, then the advantage of our domains is small. On the other hand, if a particular program and initial goal have the property that arguments are either partially instantiated or free at most program points, then a ground/non-ground domain is almost useless.

For our first experiments we used the analyser of [6]. Table 1 compares the analysis times using a ground/non-ground domain assuming a (1) ground input pattern and a (2) non-ground input pattern; using our domains assuming a (3) ground input pattern and a (4) non-ground input pattern. The figures are normalised wrt. (1). The actual times were mostly in the order of less than a second.

E.g. consider `append`. Assuming an initial goal where the first two arguments are ground, our analysis (3) infers essentially the same results obtained with a ground/non-ground domain (1). Assuming an initial goal where the first two arguments are terminated but not ground lists, our analysis (4) infers useful information (`append` is answered with the third argument being a terminated list) whereas a ground/non-ground domain (2) does not. Similar considerations apply for the other examples, except for `quicksort`, where assuming a non-ground input pattern is never very sensible, independently of the domain.
We will have to conduct more experiments. One experience was that often quite large sets of possible call and answer patterns are enumerated in the course of our analysis. We expect that much is to be gained by finding ways of representing sets of patterns more concisely.

7 Conclusion

We have presented a general domain construction for mode analysis of typed logic programs. The program's type declarations are used to construct abstract domains. Each abstract term in an abstract domain characterises the degree of instantiation of concrete terms with the same type. The *abstraction*, in the sense of "reduction to the essential", lies in characterising two properties of a term: *termination* and *instantiation of subterms*. To ensure that the abstract domains are finite, we define a hierarchy on the types and approximate all subterms that occur at the same level in this hierarchy by a single abstract term whose size is fixed.

The abstraction generalises the concept of nil-termination for lists. The `Nests` example demonstrates that this generalisation is non-trivial. However, our construction is still modular in that, for example, the abstract domain for lists is not affected by the use of lists as arguments in functions of other types. For arbitrary user-defined types, an analysis using our abstract domains is obviously more precise than previous approaches because for such types, only the modes *ground*, *non-ground*, and *free* had been considered.

The formalism that describes the construction of an abstract domain for arbitrary types which is precise and which meets the modularity requirement is, without doubt, complex. Nevertheless this formalism yields abstract domains for the common examples (lists, trees, difference lists) that are comparable to the domains used in an analysis with a fixed number of "standard" types. Thus the domains and the operation defined on the abstract terms (computing the abstract dependencies) turn out to be much simpler than the general definitions suggest.

Admittedly our work is not as general as was originally intended, which is apparent by the Simple Range Condition and the Reflexive Condition. In a study of existing Gödel programs it was noted that these conditions were never violated in practice. By imposing these conditions, the construction and the proofs of essential results were considerably simplified.

Related work

This work was strongly motivated by [3]. One primary aim had been to map those ideas developed for Prolog to typed logic programming languages. Notably the title says that *type* dependencies are derived; it does not mention modes. A type in this context is merely a set of terms, and type analysis, i.e. inferring that an argument is instantiated to a term *of a certain type*, is inseparable from mode analysis. The analysis must account for incorrectly typed terms. E.g. as it cannot be assumed that a variable tail of a list will be bound to a list, a list whose tail is uninstantiated is always abstracted as `any`. Thus it is not even possible to capture that such a list is at least *partially* instantiated.

The imprecision of only distinguishing ground and non-ground arguments in mode analysis is pointed out in [1]. They consider untyped languages and specify types by *set expressions*. Set expressions are a powerful formalism which is used for directional type checking, i.e. checking that a set of call patterns for a predicate corresponds to a particular set of answer patterns.

Mercury [14] requires the user to declare the modes of predicates. The precision with which the instantiation of an argument can be characterised is similar to the precision of our approach, although there are differences because of the different application. In particular, their mode declarations specify things such as that the elements of a list are definitely free (not even aliased!), which presumably is needed for optimising unification. In the current implementation however, they only distinguish input (ground) arguments and output (free) arguments.

Future work

The implementation needs to be improved and an extended series of experiments should be conducted to judge both the cost and the degree of precision obtained in practice.

The performance of abstract interpretation analyses depends not only on the complexity of the abstract domains but also on the techniques used to evaluate the abstract program. We will have to investigate how these techniques (shown for Prolog e.g. in [4]) translate to Gödel or other typed languages. During our implementation work we found that analysing *typed* languages adds some new aspects: Should the abstract *domain* for each type correspond to a type in the abstract program? Should it be ensured that no pattern is inferred by the abstract program whose concretisation violates the type system? Does such a restriction impede efficiency by limiting the use of non-ground structures [4] in the implementation of the analysis?

Our abstract domains may also be useful to specify delay declarations. In Gödel, it can be declared that a predicate delays until an argument is ground or at least non-variable. We found however, that these declarations do not describe the behaviour of the Gödel system predicates very precisely. Typically, the degree of instantiation required for a predicate to run safely without delaying can be precisely specified by an abstract term in our domains.

Another question is how the precision of our domains can be exploited for determinacy analysis. So far the domains are constructed taking the declarations as input. It would be interesting to investigate whether there is a general way to construct domains that also consider the program clauses and that are just precise enough to capture the information which is necessary to detect determinacy.

Acknowledgements

We thank Jonathan Martin, Tony Bowers, Bart Demoen, and Fergus Henderson for helpful discussions and comments, and Andrew Heaton for providing the software of [6] for testing our domains. 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] T. Bowers. Personal communication on the common use of polymorphism. September 1996.

[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. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Constraint logic programming with dynamic scheduling: A semantics based on close operators. April 1996.

[6] A.J. Heaton, P.M. Hill, and A.M. King. Using Pos to analyse logic programs with delay. Technical Report 97.11, School of Computer Studies, University of Leeds, 1997.

[7] F. Henderson. Personal communication on declarations in Mercury. November 1996.

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

[9] P.M. Hill and A.M. King. Determinacy and determinacy analysis. Technical Report 95.13, School of Computer Studies, 1995. accepted for publication in the Journal of Programming Languages.

[10] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[11] K. Marriott, M. García de la Banda, and M. Hermenegildo. Analyzing logic programs with dynamic scheduling. In *POPL'94*, pages 240–253. ACM Press, 1994.

[12] K. Marriott and H. Søndergaard. Precise and efficient groundness analysis. *ACM Lett. Program. Lang. Syst.*, 13(2–3):181–196, 1993.

[13] R. A. O'Keefe. *The Craft of Prolog.* MIT Press, 1990.

[14] 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.