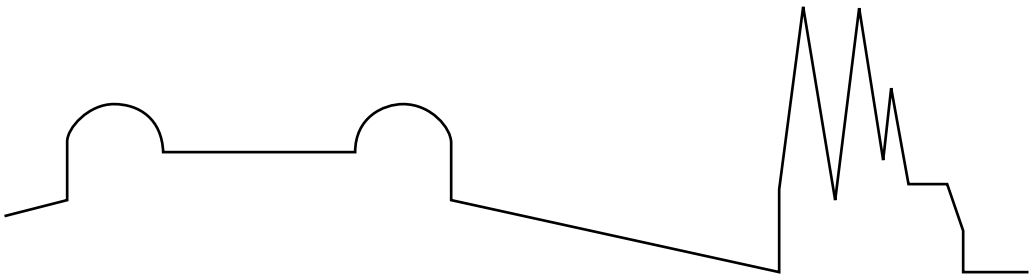
UPPSALA THESES IN COMPUTING SCIENCE
No. 23/95

IGOR: A tool for developing abstract domains for Prolog analyzers

Magnus Nordin



Thesis for the Degree of
Licentiate of Philosophy

ISSN 0283-359X

UPMAIL
Computing Science Department
Uppsala University
Box 311
S-751 05 UPPSALA
Sweden

Abstract

Abstract interpretation is a method that provides a semantic approach to program analysis. A non-standard abstract semantics is used to simulate the execution of the program to be analyzed. The result of the abstract interpretation is an approximate description of the execution of the program. These approximate descriptions are elements of an *abstract domain* and the definition of this abstract domain decides what information can be gained during the analysis. We describe a tool, IGOR, for implementing, testing, modifying, and evaluating abstract domains for analysis of Prolog programs. A high-level specification language is used for specifying abstract domains that are compiled into Prolog and interfaced with a fixpoint engine to make up a complete analyzer. The compiler automatically generates code for basic domain operations from special domain type definitions. These definition are also used for combining and reducing domains. The language provides primitives, such as set and lattice operations, and a concise method for specifying abstract interpretation of built-in predicates. We evaluate the tool and show that the high-level specifications are close to an order of magnitude less voluminous than the corresponding Prolog code and that the execution speed of the generated code is close to that of hand-written analyzers.

Contents

Chapter 1

Introduction

Program analysis is increasingly being used for optimization of programming language implementations. Abstract interpretation [8] is a method that provides a semantic approach to program analysis. A non-standard abstract semantics is used to simulate the execution of the program to be analyzed. The result of the abstract interpretation is an approximate description of the execution of the program. These approximate descriptions are expressed as elements of an *abstract domain*. The definition of the abstract domain decides what information can be collected during an analysis.

However, the engineering effort required to develop abstract interpreters and abstract domains often limits the pace in which new or improved analyses can be tested and evaluated. For example, during the development of a compiler in a Prolog system it was found that altering or extending the implementation of a moderately complex (120 Kb of Prolog code) analyzer took more than one person-week even for relatively straightforward changes [22]. Major redesigns or extensions would probably require complete re-implementation, a task of several person-months.

To improve upon this situation, we have designed and implemented a tool called Igor that, given a high-level specification of the analysis domain, generates large parts of the analyzer automatically. The performance of the automatically-generated code is close or equivalent to hand-written code.

In our experience, the tool greatly simplifies implementation, debugging and evaluation of data-flow analyzers. We have used it for designing new analyses, for combining analyses, and for implementing analyses described in the literature (while implementing several well-known analyses, we found minor errors in some of their specifications—an indication, perhaps, that a tool like this could be useful also to others).

Domain specifications are written in a first-order, statically typed strict functional language. This language has operations for manipulating sets and lattices, for projecting domains, and for combining domains. Specifications are compiled to

Prolog code and optionally linked with an analysis framework based on Getzinger's algorithm [15]. There is a completely customizable interface to the automatically-generated domain code for users that want to provide their own fixpoint engines. The system provides support for concise specification of built-in operations and for communicating analysis results to the subsequent phases of the compiler.

1.1 ORGANIZATION OF THE THESIS

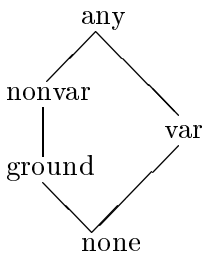
Chapter 2 gives an introductory language summary and presents some simple examples of what can be done with IGOR. The kernel of the thesis is presented in chapter 3 which gives the definition of the types and functions of IGOR. Chapter 4 handles the pragmatic aspects of the system and in chapter 5 an evaluation of the system is given. Chapter 6 and 7 close the thesis with related work, discussions, and the conclusion. Several appendices provide examples and more detailed information on subjects that are peripheral to the thesis.

Language Summary

2.1 A SIMPLE EXAMPLE

The user specifies domain *types*. Several kinds of domain types are supported: sets ordered by inclusion, product domains, atomic function domains, finite lattices, recursive domains, and disjunctive domains. When a type definition is compiled, the tool generates Prolog code for the meet, join, and comparison operations on that type. It also provides handles to the top and bottom elements of the domain.

Consider the standard domain for mode analysis of Prolog:



The values represent the mode of variables: *any* — no information, *nonvar* — not a free variable, *var* — a free variable, *ground* — bound to a ground value.

This domain is defined, by listing the chains of the lattice, with the declaration

```
type mode => lattice([[any,nonvar,ground,none],
                      [any,var,none]])
```

The compiler generates code for the operations

`mode_top`	$(\top)$
`mode_bot`	$(\bot)$
`mode_meet(A,B)`	$(A \sqcap B)$
`mode_join(A,B)`	$(A \sqcup B)$
`mode_leq(A,B)`	$(A \sqsubseteq B)$

where `A` and `B` are lattice elements. The user can override these operations with his own if needed. These operations together with an abstract interpretation framework make up a complete analyzer.

The mode domain above must be combined with a domain for variable aliasing, since it is not substitution-closed [12]. We can specify an aliasing domain that is a set of sets of variables in a clause `C` as follows.

```
type aliasing(C) => set(set(variables(C))).
```

The combined mode and aliasing domain is specified as a product domain in the following way.

```
mode_map(C) =>  variables(C) -> mode.

type mode_and_alias(C) => (mode_map(C),aliasing(C)).
```

However, we need not associate alias information with the `ground` element in the mode domain. Hence we define a *projection* that expresses this fact:

```
mode_and_alias_proj((ModeMap, Aliasing)) =>
  (ModeMap,
   {{X | X <- P, ModeMap @ X \= ground} | P <- Aliasing} \ {{}}
  ).
```

This projection removes all ground variables from the aliasing components of the `mode_and_alias` domain.

2.2 LANGUAGE FEATURES

Besides basic lattice operations, the language supports several features that are useful when specifying abstract domains.

Set expressions

The language includes expressions for traversing sets, mapping functions on sets, and universal or existential quantification over set elements. In particular, the use of set expressions allows concise definitions of aliasing properties.

Example. Consider the set expression

```
{{f(X) | X <- SubSet, p(X)} | SubSet <- Set}
```

It maps the function `f/1` on all elements `X`, satisfying property `p`, drawn from the set of sets `Set`. □

Example. Consider the existential quantification

```
exists(X <- S1, is_subset(X, S2) /\ X \= {})
```

It checks the existence of an element, `X`, of set `S1`, which is a nonempty subset of set `S2`. □

Set expressions are compiled into (possibly nested) loops traversing the sets.

Sets of known cardinality are implemented as bit vectors. The type-checker determines if a set can be represented as a bit vector. Measurements show that the efficiency of using bit vectors or not depends on which set operations the specified domain primarily relies on. Compared to an ordered list representation, the bit vector representation gives more efficient union, intersection, and member operations, at the price of a somewhat higher cost for set traversal.

Projections

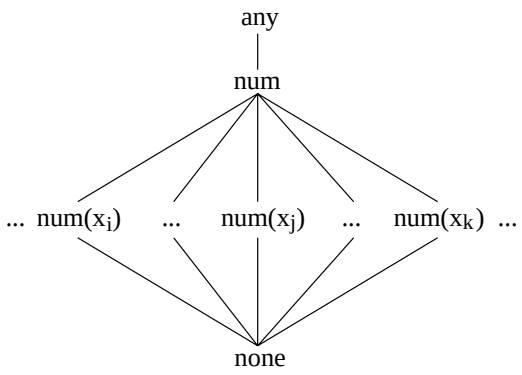
Projections make it possible to remove redundant elements from a domain, or to bound an infinite domain, by mapping multiple domain elements to a single element. Conversion functions are automatically inserted into the generated code as required.

Example. Assume that a projection `mode_proj` is defined for the domain `mode`. Then `mode_proj(mode_join(A,B))` will replace the original `mode_join(A,B)` operation. □

Attributed domains

It is sometimes convenient to be able to include references to untyped data in domain declarations. Such data are called *attributes* and the domains they occur in are called *attributed domains*.

Example. Consider the domain



Here the x's are attributes, representing the integers. The domain, call it `numbers`, can be specified by the declaration

```
type numbers => lattice([[any,num,num(X),none]])
```

Here it is convenient to treat the integers as attributes, since we have no way of declaring an infinite, unordered set in the language. □

Recursive and disjunctive domains

Some domains, e.g. domains for keeping track of the structure of compound terms, can be described by *recursive domains* in the language. These domains are infinite in size. Some elements of recursive domains are infinite trees whose nodes may contain descriptions of functors and argument types. To ensure termination of the analysis, the user must bound the depth and width of the tree by applying projections in the analysis.

Example. Assume that we are interested in tracking non-variable terms in general, and compound terms in particular. Furthermore, for arguments of the compound terms we are interested in recursively tracking non-variables and compound terms. This can be achieved with the attributed recursive domain

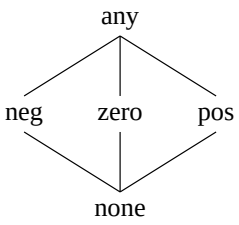
```
type term => lattice([[any,nonvar,str(F,list(term)),none]])
```

Here `F` is an attribute representing the functors of compound terms. An example of an element of this domain is `str(f,[nonvar,str(g,[any])])`, representing all terms `f(A,g(B))` where `A` is a non-variable and `B` is any term. □

The joining of domain elements often results in loss of precision. One way of remedying some of this loss is to allow disjunctive elements in the domain. IGOR supports this by *disjunctive domains.*

Example. Consider a simple domain for tracking whether numbers are negative, zero, or positive:

```
type sign => lattice([[any,neg,none],
                      [any,zero,none]
                      [any,pos,none]]).
```



We can make this a disjunctive domain by declaring it as:

```
type sign => disj_lattice([[any,neg,none],
                           [any,zero,none]
                           [any,pos,none]]).
```

Now we can express that an element is, e.g., either positive or zero: `or({zero,pos})`. Elements like `or({any,pos})` will never be explicitly created as one disjunct (`any`) is greater than all the other disjuncts (`pos`). □

Recursive and disjunctive domains can be combined to form *recursive disjunctive domains.*

Example. Consider again the recursive domain discussed above.

```
type term => lattice([[any,nonvar,str(F,list(term)),none]])
```

With this domain we cannot keep track of structures with different functors. For example, the join of `str(f,[nonvar])` and `str(g,[nonvar])` is `nonvar`. However, we can keep track of different functors by making the domain disjunctive:

```
type term => disj_lattice([[any,nonvar,str(F,list(term)),none]])
```

In this domain, `or({str(f,[nonvar]),str(f,[nonvar])})` is the join of `str(f,[nonvar])` and `str(g,[nonvar])`. □

Pragmatics

A tool like IGOR must be able to handle realistically-sized programs, not just small toy benchmarks. This requires support for various engeneering issues in the development of analyzers:

- *Normalization.* Most analyses assume that programs are written on a normalized form. IGOR provides automatic normalization, with several options adjustable by the user.
- *Inspecting and decomposing programs.* An analyzer often needs to inspect or decompose clauses and procedures in different ways. IGOR provides an extensive library to support this.
- *Built-in operations.* Specifying the built-in operations of the source language is often a tedious process. For example, SICStus Prolog provides more than 250 built-in operations, most of which are irrelevant to compiler analyses but which still require handling by the abstract domain in order to analyze real programs. IGOR provides support for concisely specifying the effects of collections of built-in operations.
- *Annotated output.* Users can have widely different uses of their analyses. Each use might require a different output format. IGOR annotates the program in a way that can be directed by the user. Optionally, the annotated program can be prettyprinted.

Language Definition

One of the motivations behind this work was to enable the use of formal domain specifications, or something very close to them, as the source code of the implementation. As these formal specifications are mostly functional, the language is designed as a statically typed first order functional language with some higher order extensions. The primary data type is sets.

Special domain types are used to define the abstract domains needed for a specific analysis. These type declarations are used to automatically generate code for the most frequent primitive operations associated with domains. Examples of such operations are: generate top and bottom elements, compute least upper bound and greatest lower bound, and compare domain elements.

The abstract operations, e.g., abstract unification, which approximates the concrete operations of logic programs, are defined as functions. Certain functions serve as interface functions between the abstract domain operations and the analyzer framework. A special language construct facilitates compact definition of the abstract behaviour of built-in predicates.

3.1 DOMAIN TYPES

There are primitive and complex domain types. Primitive types represent the sets used as a basis for complex types. The primitive types are not used in isolation. Complex types represent domains and have a number of domain functions associated with them. The syntax of type declarations is described in Table 3.1.

The syntax of type declarations

N ::= integers | integer variables.	
C ::= constants.	
V ::= variables.	
T ::= `type` f(V, ..., V) `=>` D.	*Type definition*
P ::= `index`(N)	*Primitive domains*
| `range`(N, N)	
| `elements`([C, ..., C])	
| `variables`(V).	
D ::= `flat`(P)	*Complex domains*
| `set`(D)	
| `invset`(D)	
| (P `->` D)	
| (D, ..., D)	
| `ntuple`(D, N)	
| `list`(D)	
| `lattice`([L, ..., L])	
| `disj_lattice`([L, ..., L]).	
L ::= [E, ..., E].	*Chains of lattice elements*
E ::= f(R, ..., R) | C.	*Lattice elements*
R ::= V | D.	*Attributes or recursive elements*

Table 3.1: The syntax of type declarations in BNF. We use "f(...)" to represent compound terms and "[...]" to represent lists.

Types are specified with the declaration

`type` *type_head* `=>` *type_body*`.`

The type head can contain arguments, typically an object program clause or clause head. The type body defines the type by combining primitive and complex types. An example:

```
type aliasing(Clause) => set(set(variables(Clause))).
```

The type defined is called `aliasing` and the elements of this type are sets of sets of variables of a term, `Clause`. `set` is a predefined complex domain type and `variables` is a predefined primitive domain type.

Primitive domain types

The role of the primitive domain types is to define the most basic elements of a domain. The following primitive types are used by the complex types to build domains. All primitive types are sets.

- `index(`N`)` the set of integers between 1 and N.
- `range(`M`, `N`)` the set of integers between M and N, inclusive.
- `elements([`$e_1, ..., e_n$`])` the set of constants e_1 to e_n.
- `variables(`T`)` the set of variables in term T.

Complex domain types

Complex domain types use other, primitive or complex, domain types as subtypes. All complex domain types have associated domain operations (see Section 3.3) generated by the IGOR compiler.

The semantics of the automatically generated domain operations are defined below. When the set of elements of the domain is enumerable, an enumeration function is defined. We will write e and f for domain elements below.

- `flat(`T`)`

This is the flat domain constructed by adding top ($\top$) and bottom ($\bot$) elements to sets of type T. The elements of `flat(`T`)` are thus $T \cup \{\top, \bot\}$.

$$e \sqsubseteq f \quad \Longleftrightarrow \quad e = f \text{ or } f = \top \text{ or } e = \bot$$

$$e \sqcap f \quad = \quad \begin{cases} e, & \text{if } f = e \text{ or } f = \top \\ f, & \text{if } e = \top \\ \bot, & \text{otherwise} \end{cases}$$

$$e \sqcup f \quad = \quad \begin{cases} e, & \text{if } f = e \text{ or } f = \bot \\ f, & \text{if } e = \bot \\ \top, & \text{otherwise} \end{cases}$$

Example: `type t => flat(range(0,9))`

- `set(`T`)`

This is the domain constructed from the power-set of T. The elements of `set(`T`)` are the subsets of T.

$$\begin{array}{rcl} e \sqsubseteq f & \Longleftrightarrow & e \subseteq f \\ e \sqcap f & = & e \cap f \\ e \sqcup f & = & e \cup f \\ \top & = & T \\ \bot & = & \emptyset \end{array}$$

Example: `type sharing(C) => set(set(variables(C)))`

- `invset(`T`)`

This is the domain constructed from the power-set of T but with the inverse ordering of `set(`T`)`. The elements of `invset(`T`)` are the subsets of T.

$$\begin{array}{rcl} e \sqsubseteq f & \Longleftrightarrow & e \supseteq f \\ e \sqcap f & = & e \cup f \\ e \sqcup f & = & e \cap f \\ \top & = & \emptyset \\ \bot & = & T \end{array}$$

Example: `type free_vars(C) => invset(variables(C))`

- `(`T_1 `->` T_2`)`

This is the atomic function domain constructed from the atomic subtype T_1 and a complex domain type T_2. In the implementation all enumerable subtypes, i.e.

those types with an associated *type*_`elements` function, are considered atomic. The elements of (`(`T_1 `->` T_2`)`) are not enumerable. For all $x \in T_1$,

$$\begin{array}{rcl} e \sqsubseteq f & \iff & e(x) \sqsubseteq_{T_2} f(x) \\ (e \sqcap f)(x) & = & e(x) \sqcap_{T_2} f(x) \\ (e \sqcup f)(x) & = & e(x) \sqcup_{T_2} f(x) \\ \top(x) & = & e(x) \mapsto \top_{T_2} \\ \bot(x) & = & e(x) \mapsto \bot_{T_2} \end{array}$$

Example: `type mode_map(C) => variables(C) -> mode`

- (`(`$T_1, ..., T_n$`)`)

This is the product domain $T_1 \times ... \times T_n$.

$$\begin{array}{rcl} e \sqsubseteq f & \iff & e_{T_i} \sqsubseteq_{T_i} f_{T_i} \text{ for all } i \in \{1, n\} \\ e \sqcap f & = & (e_{T_1} \sqcap_{T_1} f_{T_1}, ..., e_{T_n} \sqcap_{T_n} f_{T_n}) \\ e \sqcup f & = & (e_{T_1} \sqcup_{T_1} f_{T_1}, ..., e_{T_n} \sqcup_{T_n} f_{T_n}) \\ \top & = & (\top_{T_1}, ..., \top_{T_n}) \\ \bot & = & (\bot_{T_1}, ..., \bot_{T_n}) \end{array}$$

Example: `type desc(C) => (mode_map(C), aliasing(C))`

- `ntuple(`T`, `N`)`

This is the product domain $\overbrace{T \times ... \times T}^{N \text{ times}}$. N must be an integer.

$$\begin{array}{rcl} e \sqsubseteq f & \iff & e_i \sqsubseteq_T f_i \text{ for all } i \in \{1, n\} \\ e \sqcap f & = & (e_1 \sqcap_T f_1, ..., e_N \sqcap_T f_N) \\ e \sqcup f & = & (e_1 \sqcup_T f_1, ..., e_N \sqcup_T f_N) \\ \top & = & (\top_T, ..., \top_T) \\ \bot & = & (\bot_T, ..., \bot_T) \end{array}$$

Example: `type desc(C) => ntuple(mode, arity(C))`

• `list(`T`)`

This type is included since it is useful when defining recursive lattices (see Section 3.2). `list(`T`)` is similar to `set(`T`)` but with ordered and possibly duplicated elements, i.e., lists. `list(`T`)` defines several domains as elements of `list(`T`)` must have the same length to be comparable. Consequently, there is one list domain for every set of lists of length $n \geq 0$. We denote the empty list, $n = 0$, as nil. The elements of `list(`T`)` are not enumerable.

$$\begin{array}{rcl} e \sqsubseteq f & \Longleftrightarrow & \text{first}(e) \sqsubseteq_T \text{first}(f) \wedge \text{rest}(e) \sqsubseteq \text{rest}(f) \\ e \sqcap f & = & \text{cons}(\text{first}(e) \sqcap_T \text{first}(f), \text{rest}(e) \sqcap \text{rest}(f)) \\ e \sqcup f & = & \text{cons}(\text{first}(e) \sqcup_T \text{first}(f), \text{rest}(e) \sqcup \text{rest}(f)) \\ \top_n & = & \text{cons}(\top_T, \top_{n-1}) \text{ where } \top_0 = \text{nil} \\ \bot_n & = & \text{cons}(\bot_T, \bot_{n-1}) \text{ where } \bot_0 = \text{nil} \end{array}$$

The lists e and f have the same length. Additionally, nil $\sqsubseteq$ nil, nil $\sqcap$ nil = nil, and nil $\sqcup$ nil = nil. An example of the use of this type is in the section on recursive lattices (see page 16).

3.2 LATTICE TYPES

The domain operations of explicitly defined lattices are tedious and sometimes difficult to implement. For these reasons IGOR supplies several lattice types of various complexity.

• `lattice(`$[[e_{1,1}, ..., e_{1,n_1}], ..., [e_{k,1}, ..., e_{k,n_k}]]$`)`

This is the set of elements $e_{x,y}$ ordered by the chains $e_{1,1} \sqsupset ... \sqsupset e_{1,n_1}$ through $e_{k,1} \sqsupset ... \sqsupset e_{k,n_k}$. The chains should specify the order in such a way that unique top and bottom elements exist. Each relation (arc) of the lattice need only be specified in one of the chains. The elements of the lattice type are the elements in the chains.

Example. The definition

```
type mode => lattice([[any,nonvar,ground,none],[any,var,none]]).
```

defines the lattice shown in Figure 3.1. □

Attributed lattices

It is possible to attach one or more *attributes* to elements of a lattice. This is specified as $e(A_1, ..., A_n)$ where $A_1, ..., A_n$ are attribute variables, representing arbitrary

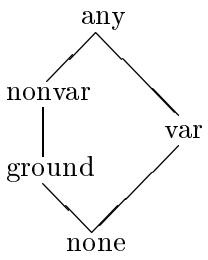


Figure 3.1: An example of a simple lattice domain.

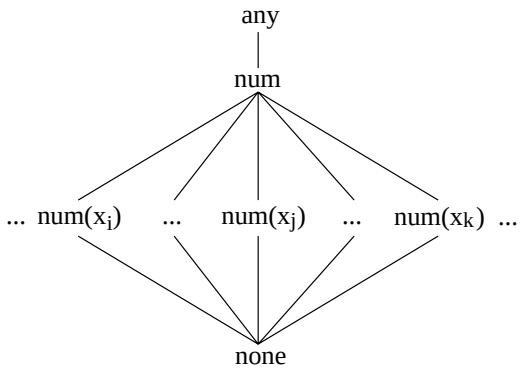


Figure 3.2: An example of an attributed lattice.

terms that are neither ordered nor typed. e is the functor of the element. An example: the specification $e(A)$ attaches one attribute A to e. The elements $e(1)$, $e(foo)$, and $e((bar, 42))$ are well-formed elements of a lattice type that includes this element specification.

The motivation behind attributed domain elements is that it is often desirable to store arbitrary information about the objects being analyzed in the domain elements.

An example of an attributed domain is shown in Figure 3.2. This is the lattice defined by the domain type definition `lattice([[any,num,num(X),none]])`. The x's are attributes, representing the integers.

Attributed domain elements are ordered according to their placement in the chains of the lattice specification. To unclutter the presentation of the rules defining

the domain operations, we restrict the rules to elements with one attribute only. Generalization to any number of attributes follows easily. To define the semantics of domain operations, we will need a few definitions:

Definition: Let D be an ordered set and let $x, y, z \in D$. Then x is **covered by** y, if and only if $x \sqsubset y$ and $x \sqsubseteq z \sqsubset y$ implies $z = x$. We write this relation as $x \prec y$.

Definition: Let D be an ordered set and let $x, y, z \in D$. Then x is **join-irreducible** if and only if $x \neq \bot$ and $(\forall y, z \in D)(y \sqsubset x \wedge z \sqsubset x$ implies $y \sqcup z \sqsubset x)$. Meet-irreducibility is defined conversely.

Definition: Let a and b be lattice elements then a and b are **non-comparable**, written $a \parallel b$, if and only if $a \not\sqsubseteq b \wedge b \not\sqsubseteq a$.

Let D be a lattice, $e(A)$ an attributed element, A an attribute variable, a and b arbitrary attributes in D. Attributed elements with different functors are operated on according to the normal lattice order. Attributed elements with identical functors are operated on according to the following rules:

$$
\begin{array}{lcl}
e(a) \sqsubseteq e(b) & \Longleftrightarrow & a = b \\
e(a) \sqcap e(b) & = & \begin{cases} g \text{ where } g \in D \text{ and } g \prec e(A), & \text{if } a \neq b \\ e(a), & \text{otherwise} \end{cases} \\
e(a) \sqcup e(b) & = & \begin{cases} g \text{ where } g \in D \text{ and } e(A) \prec g, & \text{if } a \neq b \\ e(a), & \text{otherwise} \end{cases}
\end{array}
$$

Example. Using the domain shown in Figure 3.2, $num(1) \sqcup num(2) = num$ and $num(1) \sqcap num(2) = none$. □

To ensure the existence of a unique element g for the meet- and join-operations, an attributed element must be join- and meet-irreducible. This constraint also prohibits the joining or meeting of two elements into an attributed element, in which case it would be impossible to calculate the attribute of the result.

For an example of the use of an attributed lattice, see the the definition of domain `str` in Appendix C.5.

Recursively defined lattices

So far we have only discussed finite domains. Some domains are potentially infinitely wide and unbounded in depth. These domains, which are useful for keeping

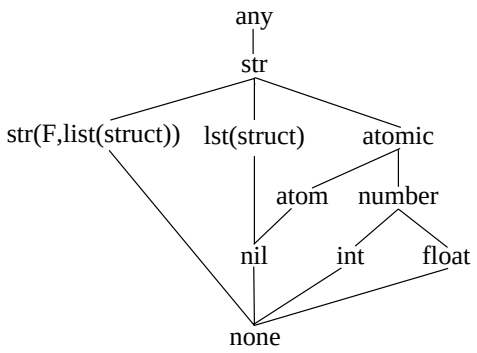


Figure 3.3: An example of an attributed recursive lattice.

track of the structure of compound terms, can be specified with the recursive lattice type. The elements of recursive lattices are trees whose nodes may contain, e.g., descriptions of functors and argument types.

Recursive elements of lattice domains are specified as $e(T_1, ..., T_n)$ where $T_1, ..., T_n$ are types. These subtypes may be identical with the current lattice being defined, i.e. the domains can be directly or indirectly recursive. Unless they have identical functors, recursively defined elements are ordered in the same way as non-recursive elements, i.e., according the their placement in the order chains.

Example. The domain in Figure 3.3 is specified by the following declaration:

```
type struct => lattice([[any,str,str(F,list(struct)),none],
                        [str,lst(struct),nil]
                        [str,atomic,atom,nil,none],
                        [atomic,number,int,none],
                        [number,float,none]]).
```

□

Let $e(T)$ be a recursive domain element, x and y be elements of type T.

The domain operations are defined as:

$$\begin{array}{rcl} e(x) \sqsubseteq e(y) & \Longleftrightarrow & x \sqsubseteq_T y \\ e(x) \sqcap e(y) & = & e(x \sqcap_T y) \\ e(x) \sqcup e(y) & = & e(x \sqcup_T y) \end{array}$$

Example. In the domain of Figure 3.3, $lst(atom) \sqcup lst(number) = lst(atomic)$ and $atomic \sqcap lst(any) = nil$. □

If a recursive element is attributed, the rules for attributed elements and recursive elements are combined. Rules for attributed elements have the highest precedence.

Let A be an arbitrary attribute, $e(A,T)$ be an attributed recursive element, a and b be attributes, x and y arguments of type T. Then the domain operations are defined as:

$$
\begin{array}{lcl}
e(a,x) \sqsubseteq e(b,y) & \Longleftrightarrow & a = b \wedge x \sqsubseteq_T y \\
e(a,x) \sqcap e(b,y) & = & \begin{cases} g \text{ where } g \in D \wedge g \prec e(A,T), & a \neq b \\ e(a, x \sqcap_T y), & \text{otherwise} \end{cases} \\
e(a,x) \sqcup e(b,y) & = & \begin{cases} g \text{ where } g \in D \wedge e(A,T) \prec g, & a \neq b \\ e(a, x \sqcup_T y), & \text{otherwise} \end{cases}
\end{array}
$$

Example. Again, using the same domain, we have $str(a,[any]) \sqcup str(b,[int]) = str$ and $str(a,[any]) \sqcap str(a,[int]) = str(a,[int])$. □

The elements of attributed and recursively defined lattices are not enumerable.

Disjunctive lattice types

As can be seen in the previous example, operations sometimes result in the loss of information. When joining the two elements with different attributes, a and b, we end up with the general *str*-element. To remedy this IGOR supports disjunctive lattices. Let γ be the 'concretization' function that maps abstract values to terms. A disjunctive lattice D is formed from a lattice D_0 by introducing extra elements. A new element $or(\{a_1, \ldots, a_k\})$, where $a_1, \ldots, a_k \in D_0$, is introduced to D, if it is not known that $\gamma(a_1 \sqcup \cdots \sqcup a_k) = \gamma(a_1) \cup \cdots \cup \gamma(a_k)$. This equation can, in general, only be verified by the user. However there are special cases which are automatically detectable. For example, the system does not add a new element to the domain if there is one disjunct, a_n, in $a_1, \ldots, a_k$ that is greater than all the other members of this set of disjuncts (in this case only a_n is used.)

- `disj_lattice(`$[[e_{1,1}, ..., e_{1,n_1}], ..., [e_{k,1}, ..., e_{k,n_k}]]$`)`

This is an extension to the lattice type. In addition to all features of the lattice type, disjunctive elements are added to the lattice. The disjunctive elements are

represented as $or(S)$, where S is a set of disjuncts. We call such a set a disjunctive set. The expression $or(\{x_1, ..., x_n\})$ equals $x_1 \vee ... \vee x_n$. These elements are not explicitly defined in the type definition, but are constructed by the join-operation of disjunctive lattices. An example: if a and b are non-comparable, then $a \sqcup b = or(\{a, b\})$. In addition to the rules of the lattice type, a number of new rules are required for the operations on disjunctive types. These new rules take precedence over the previous lattice rules.

Example. The disjunctive version of the domain in Figure 3.3 is specified by the following declaration:

```
type struct => disj_lattice([[any,str,str(F,list(struct)),none],
                             [str,lst(struct),nil]
                             [str,atomic,atom,nil,none],
                             [atomic,number,int,none],
                             [number,float,none]]).
```

□

Let x and y elements of a disjunctive lattice, D, and let X and Y be disjunctive sets of elements of D. The domain operations of disjunctive lattice domains are:

$$\begin{aligned} or(X) \sqsubseteq or(Y) &\iff (\forall x \in X)(\exists y \in Y) x \sqsubseteq y \\ x \sqsubseteq or(Y) &\iff (\exists y \in Y) x \sqsubseteq y \\ or(X) \sqsubseteq y &\iff (\forall x \in X) x \sqsubseteq y \end{aligned}$$

Example. In the domain `struct` defined above $or(\{nil, int\}) \sqsubseteq str$ holds. □

$$\begin{aligned} or(X) \sqcap or(Y) &= or(\bigcup_{x \in X} (Y \ominus x)) \\ or(X) \sqcap y &= or(X \ominus y) \\ x \sqcap or(Y) &= or(Y \ominus x) \end{aligned}$$

where

$$Y \ominus x = \{x \sqcap y \neq \bot \mid y \in Y\}$$

The $\ominus$-operator performs deletion of elements from disjunctive sets. It collects the results of meet-operations between an element, x, and all elements of a disjunctive set Y. It discards any $\bot$-elements.

Example. In the domain `struct` the meet of the elements $or(\{int, lst(str)\})$ and $or(\{lst(atom), atomic\})$ equals $or(\{int, lst(atom), nil\})$. The meet of the elements $or(\{int, lst(str)\})$ and $str(a, [any])$ equals $or(\emptyset)$ which is equivalent to $\bot$. □

Let $a_1, ..., a_n$, $b_1, ..., b_n$, $n \geq 0$ be attributes, let $e_1, ..., e_m$, $f_1, ..., f_m$, $m \geq 0$, be elements of any complex domain type, and let $x = e(a_1, ..., a_n, e_1, ..., e_m)$, $y = f(b_1, ..., b_n, f_1, ..., f_m)$, be elements of a disjunctive lattice, D, and let $X = \{x_1, \ldots, x_n\}$ and Y be disjunctive sets. The join operation of disjunctive lattice domains is defined as:

$$
\begin{aligned}
or(X) \sqcup or(Y) &= or(Y \oplus x_1 \oplus \ldots \oplus x_n) \\
or(X) \sqcup y &= or(X \oplus y) \\
x \sqcup or(Y) &= or(Y \oplus x) \\
x \sqcup y &= \begin{cases} x, & \text{if } y \sqsubseteq x \\ y, & \text{if } x \sqsubseteq y \\ or(\{y\} \oplus x), & \text{if } x \parallel y \end{cases}
\end{aligned}
$$

The $\oplus$-operation is the addition of an element to a disjunctive set.

$$
Y \oplus x = \begin{cases} Y, & \text{if } x = \bot \\ \{x\}, & \text{if } \forall y \in Y, y \sqsubseteq x \\ Y \backslash \{y\} \cup \{e(a_1, ..., a_n, e_1 \sqcup f_1, ..., e_m \sqcup f_m)\}, & \text{if } \exists y \in Y \text{ s.t. } x \text{ matches } y \\ Y \cup \{x\}, & \text{otherwise} \end{cases}
$$

An explanation of the definition of $\oplus$ follows:

1. The first case eliminates any $\bot$-elements from a disjunction.
2. The second case replaces a disjunctive set with a single element if it is greater than all elements in the disjunctive set.
3. The third case, the match-case, is needed to keep the disjunctive elements as simple as possible. This cases "pushes" disjunctions down as far as possible in recursive disjunctive elements. For example, consider the expression $\{f(int)\} \oplus f(list)$; without the match-case the result is $\{f(int), f(list)\}$; with the match-case the result is $\{f(or(\{int, list\}))\}$, assuming the argument of f is a recursively defined type.

4. When none of the above cases are applicable, the last case just adds an element to a disjunctive set.

The match-operation is defined as:

$$x \text{ matches } y \iff e = f \wedge a_1 = b_1 \wedge ... \wedge a_n = b_n,$$

where

$$x = e(a_1, ..., a_n, e_1, ..., e_m)$$
$$y = f(b_1, ..., b_n, f_1, ..., f_m)$$

i.e., the functor, arity, and all top-level[1] attributes of x and y are identical.

Example. Using the same lattice domain again, $nil \sqcup int = or(\{nil, int\})$ and $str(a, [int]) \sqcup or(\{atom, str(a, [float])\}) = or(\{atom, str(a, [or(\{int, float\})])\})$. Note that the disjunction $or(\{int, float\})$ is equivalent to the *number*-element. However, this is not detected by the system. □

If a disjunctive set is reduced to one element or less then the following rules hold:

$$\begin{aligned} or(\{x\}) &= x \\ or(\emptyset) &= \bot \end{aligned}$$

In addition to the domain operations generated for the disjunctive lattice, operations are generated for the corresponding lattice without disjunctions. These operations are prefixed with *type*`_basic`; the join-operation is called *type*`_basic_join`. This is to facilitate bounding or reduction of disjunctive elements with projections, e.g.:[2]

```
type_proj(E) =>
  ( is_disjunctive(E) /\ #(disjuncts(E)) > 4 ->
    type_basic_join(disjuncts(E))
  ; E ).
```

[1] As opposed to attributes occurring in recursive arguments of x and y.

[2] This is a simplification. We must make sure that there are no disjunctive elements occuring in the arguments of the members of `disjuncts(E)`. See the example domain `str.ad` for a more realistic example.

Note that a member of a disjunctive set can never be a disjunctive element itself. It is possible, however, that a recursively defined member of a disjunctive set contains another disjunctive element in one of its arguments. In other words $or(\{or(\{a,b\}),c\})$ is not a possible element, but $or(\{f(or(\{a,b\})),c\})$ is.

Notes on type combinations

Product domains, lists, and recursive lattices require that their subtypes also be complex types in order to be able to construct their domain operations. For example, to be able to construct the join-operation of the product type $c = (a, b)$, a and b must have join-operations, i.e. be complex types, since $x \sqcup_c y = (x \sqcup_a y, x \sqcup_b y)$. The elements of a type are only enumerable if the elements of all subtypes are enumerable.

Care must be taken by the user to avoid the creation of infinite elements, either vertically, via a self-referential domain type, or horizontally, via disjunctive sets of attributed lattice elements.

3.3 DOMAIN OPERATIONS

From these domain type definitions a set of domain operations are automatically generated by the compiler in IGOR. All domain types, except lists, atomic function domains, and lattices, but including primitive types, have the function *type*`_elements` generated. This function enumerates all elements of *type*. All complex types have the functions *type*`_leq` ($\sqsubseteq$), *type*`_meet` ($\sqcap$), *type*`_join` ($\sqcup$), *type*`_top` ($\top$), and *type*`_bot` ($\bot$) generated. The meet and join functions are supplied in both binary and unary versions. An example:

```
type mode => lattice([[any,nonvar,ground,none],[any,var,none]]).
```

results in the generation of the following functions:

```
mode_leq(E,F),
mode_meet(E,F), mode_meet(S),
mode_join(E,F), mode_join(S),
mode_top,
```

```
mode_bot.
```

As previously mentioned, it is possible to supply parameters to type declarations. These parameters are passed on to the domain functions they concern (elements, top, and bottom). For example (definite freeness):

```
type free(T) => invset(vars(T)).
```

generates the following functions:

```
free_elements(T),
free_leq(E,F),
free_meet(E,F), free_meet(S),
free_join(E,F), free_join(S),
free_top(T),
free_bot(T).
```

One use of parameters is to pass the clause or clause head to some of the domain operations, as these need to access information about the program to create certain domain elements.

It is possible to override the definitions of the automatically generated domain functions by user supplied definitions, using the same names as the automatically generated functions would have used, in the specification.

Projections

Projections makes it possible to remove redundant elements from a domain, or to bound an infinite domain, by mapping multiple domain elements to a single element. Another application of projections is to decrease analysis complexity by making the domain coarser. Whenever a projection is defined for a particular domain, all operations on the domain are 'filtered' through the projection.

A *projection* is an idempotent ($f(x) = f(f(x))$) and extensive ($x \sqsubseteq f(x)$) function. It is possible to automatically apply projections to the user defined domains. If the function *type*`_proj` is defined it will be applied to all generated functions that return domain elements. Example: if `mode_proj(E)` is defined, `mode_join(E,F)` will be generated as `mode_proj(mode_join(E,F))`, as will all other domain functions, except `mode_leq(E,F)`. There are other situations in which the user might want to apply the projection explicitly, e.g., after an abstract unification operation.

As some projections can be expensive to compute, care should be taken in using them. In some situations it is better to use a more specifically applied projection, i.e., one that is only used with explicit function applications instead of the automatic wrapping of domain operations.

The system does not check that a projection on a domain type is a projection in the strict mathematical sense. Hence, any function over a domain could be defined as a projection; a non-projection would not be very useful, though.

3.4 USER-DEFINED FUNCTIONS

Other domain operations of the abstract domain are defined in a first order functional language with pattern matching, local variable definitions, and some restricted higher order extensions.

Language fundamentals

The syntax of the language is a compromise between being close to the formal definitions of abstract domains and the convenience of using the built-in Prolog reader.

Function definitions have the form

$head_1$ `=>` $body_1$.
⋮
$head_n$ `=>` $body_n$.

The heads of the clauses must be mutually exclusive with regard to pattern matching.

The expressions that can be used for pattern matching and data construction are list separation and construction (`[Head|Tail]`), tuples (`(T1, ..., Tn)`), and the constructors used in recursive and disjunctive type definitions. The constructor {e_1`,` `...,` e_n} can be used for explicit set construction, but not for pattern matching.

Local variables are defined with `where`-expressions. The expression

$head$ `=>` $body$
    `where`
    V_1 `=` $expr_1$`,`
    ⋮
    V_n `=` $expr_n$.

will bind the variables $V_1, ..., V_n$ to the result of indicated expressions in *body*. There are no local function definitions.

Set Expressions

As most of the expressions in the abstract domain specifications work with sets, Igor supports set expressions. These expressions are based upon Turner's ZF-expressions [29] and are implemented as a variant of list comprehensions [25].

A set expression looks like:

$$\{expr \texttt{ | } V_1 \texttt{ <- } expr_1\texttt{, } \ldots\texttt{, } V_n \texttt{ <- } expr_n\texttt{, } pred\}$$

expr stands for an expression and *pred* for a boolean function. This expressions corresponds to $\{expr \mid V_1 \in expr_1 \wedge ... \wedge V_n \in expr_n \wedge pred\}$ or to the expression $\{expr \mid V_1 \in expr_1 \wedge ... \wedge V_n \in expr_n\}$, as *pred* is optional.

Example.

All $X \in S$ such that $X \sqsubseteq_{mode} ground$ is written as

```
{X | X <- S, mode_leq(X,ground)}
```

and the set of all sets $X \cup A$ where $X \in P, P \in S$ and $X \neq \emptyset$ is written as

```
{X + A | X <- P, P <- S, X \= {}}.
```

□

Another type of set expressions supported by Igor are existentially or universally quantified boolean set expressions.

Existential quantification:

`exists(`V_1 `<-` $expr_1$`, ...,` V_n `<-` $expr_n$`,` $pred$`)` corresponds to the expression: $\exists V_1 \in expr_1, ..., \exists V_n \in expr_n$ such that *pred* is true.

Example.

$(\exists X \in S)(\exists Y \in T)X \in Y$ is written as

```
exists(X <- S, Y <- T, is_member(X, Y)).
```

□

Universal quantification:

`forall(`V_1 `<-` $expr_1$`, ..., `V_n `<-` $expr_n$`,` *pred*`)` corresponds to the expression: $(\forall V_1 \in expr_1)...(\forall V_n \in expr_n)$ *pred* is true.

Example. $(\forall X \in S)X \subseteq T$ is written as

```
forall(X <- S, is_subset(X, T)).
```

□

Set expressions are compiled into code using ordered, i.e. sorted, lists or bit vectors for set representation according the inferred type of the sets traversed. Bit vectors are used when the elements of a set can be mapped to bounded integers, e.g. the set of clause variables.

Built-in functions

IGOR provides a large set of built-in functions suited for abstract domain specification. We will give a short overview of the the most important built-in functions and refer to the IGOR manual [24] for a complete description. This overview should enable the reader to understand the domain examples in Appendix C.

The most frequently used set functions, whose syntax are not self-explanatory, are: union (`+`), intersection (`*`), difference (`\`), cardinality (`#`), and the set of integers in the range N–M, which is written as `(N .. M)`. Some of these function symbols are overloaded to also represent their normal arithmetic function.

The boolean operations of the language are conjunction (`/\`), disjunction (`\/`), and negation (`not`). The conditional if-then-else-expression is on the Prolog form, i.e., (*expr*$_1$ `->` *expr*$_2$ `;` *expr*$_3$`)`. The else-branch cannot be excluded.

An analyzer often needs to inspect or decompose program clauses and terms in different ways. IGOR provides an extensive library for supporting program inspection and decomposition. Among the provided functions are `head` and `body` to access parts of clauses, `is_var`, `is_ground` and similar predicates to classify program objects, and `vars` to collect the variables of a term.

In abstract interpretation of logic programs, variable renaming is needed. This is taken care of by the function `rename` in IGOR. This function constructs an identical copy of a term, except for the variables of the term, which are replaced by new unique variables. The predicate `is_pvar` is used to recognize original program variables.

Among the remaining built-in functions are `update` which updates an element of an atomic function domain and `closure` which performs a fixpoint computation,

the closure of a function over a set. Closure of a binary function, F, over a set S is defined as $(\forall X \in S)(\forall Y \in S) \Rightarrow F(X, Y) \in S$.

Example. To compute the closure of union over a set, S, we use the expression `closure(+, 2, S)`. The second argument tells the system that it is binary union we wish to compute.

```
closure(+, 2, {{a},{b}}) = {{a},{b},{a,b}}
```
□

Pragmatics

A tool like Igor must be able to handle realistically-sized programs, not just small toy benchmarks. In this section we discuss the pragmatic features of Igor. Some of these features are supplied by the abstract interpretation framework rather than Igor.

Specification of Analysis of Prolog Built-Ins

Specifying the built-in operations of the source language is often a tedious process. For example, SICStus Prolog provides more than 250 built-in operations, most of which are irrelevant to compiler analyses but which still require handling by the abstract domain in order to analyze real programs. Igor provides support for concisely specifying the effects of collections of built-in operations.

The function `builtin` is used to specify the abstract behaviour of built-in predicates. This functions is not limited to the usual constructors for its pattern matching. In the head of `builtin` any Prolog term is considered a constructor for pattern matching. There are two forms of the function:

`builtin(`*Pattern*`, `*Desc*`) => `*body*`.`

`builtin({`$name_1$`, ..., `$name_n$`}, `*Pattern*`, `*Desc*`) => `*body*`.`

Pattern is a Prolog term matching a built-in and *Desc* is the current abstract state[1]. The first form is used to specify the analysis for a single built-in predicate. The second form can be used to specify several similar built-in predicates which are treated identically during analysis.

Example.

[1] We will use the term *descriptor* for the abstract state computed during abstract interpretations since an abstract state can include more information than the more common term *abstract substitution* implies.

```
builtin(X = Y, D) => amgu(X, Y, D).

builtin({<,>,=<,>=,=\=}, op(X,Y), D) => integer(X, integer(Y, D)). □
```

The functor of the pattern, `op(X,Y)`, in the second form is irrelevant. The `integer/2`-function in the above example returns a new version of the abstract state descriptor `D`, in which `X` and `Y` are described as integers.

The Compiler

The functional domain specification is compiled into a Prolog program ready to be interfaced with an analyzer framework. The feasibility of compiling high-level languages to Prolog has been convincingly shown by Debray [13]. There are essentially three phases in the compilation: an expansion phase that builds the code for set expressions and analysis of built-ins; a type inference phase that, if the specification is type consistent, produces a type annotated version of the specification; and a translation phase that translates the functions into Prolog code. Using the type annotations, the last phase also sees to that the correct and most efficient Prolog primitives are chosen for overloaded functions and set functions.

The type checker checks and infers the types of all expressions in the specification. It uses the Hindley/Milner type system [23] to produce an annotated version of domain specification. The type consistency of the specification is of course decided by the type checker, but the primary use of the inferred types is to decide when it is possible to use a more efficient set representation using bit vectors. This representation is used when the type checker can infer that a data object is a set of program variables. For a complete description of the type annotations see the IGOR manual [24].

The compiler checks that the chains of all defined lattice types really constitute proper lattices.

Inclusion of Prolog Predicates

It is possible to include deterministic Prolog predicates in specifications. To enable the type checker to work with these predicates it is necessary to specify the types of the included Prolog predicates. This is done with the type annotated declaration:

`prolog` *type*:*predicate*`(`*type*$_1$:X_1`, ..., `*type*$_n$:X_n`).`

Example.

```
prolog bool:is_proper_list(term:T).
```

```
is_proper_list([]).
is_proper_list([H|T]) :- ...
```

□

Included Prolog predicates use the last argument for the return value.

Presentation of Analysis Results

Users can have widely different uses for their analyses. Each use might require a different output format. The analyzer of IGOR can annotate the analyzed program with analysis results. These annotations can be specified by the user who can control what domain data should go where in the annotated version of the analyzed program.

Interface to framework

To use the abstract domain the abstract interpretation framework one must be able to access the domain operations. As the generated domain operations are translated into Prolog, they are ready to be used directly by the framework coded in Prolog.

In addition to the generated domain operations the framework uses a few interface functions to access the abstract domain. These functions are `pred_entry`, `clause_entry`, `clause_exit`, and `pred_exit` and they must be defined in the abstract domain specification. A full explanation of the interface between the framework and the abstract domain is given in Appendix A.

Evaluation

We evaluate two aspects of the system: software metrics and performance. The software metrics evaluated are the size of the domain specifications, the size of the generated code, and the time to compile specifications. The performance measures are the efficiency of the generated code compared to hand-coded implementations and the comparative efficiency of different set representations.

All measurements were made on a Sun 630 MP with a 55 MHz processor and 128 Mb of memory. The time unit is seconds. The benchmarks were interrupted if they had not completed within 1000 seconds. SICStus Prolog [5] version 2.1.9, with the *fastcode* option on, was used.

The domains used in the evaluation are `Sund`, Sundararajan's domain for freeness, sharing, and linearity [27]; `J&L`, Jacobs's and Langen's sharing domain [19]; `Str`, a simple depth-k structure domain; `Deb`, one of Debray's substitution-closed type domains [12]; `Dep`, Debray's mode and dependency domain [11]. The set of programs analyzed in the evaluation is a subset of the Berkeley benchmarks [26]. A description of the benchmarks is found in Appendix B.

Domain and compilation statistics

The size of the specifications of domains we have implemented range between 1 to 3 pages of non-commented code. These specifications include basic domain operations, abstract unification, abstract interpretation of 30 built-in predicates,

Domain	`Sund`	`J&L`	`Str`	`Deb`	`Dep`
Compilation time	11.8	6.2	15.5	8.4	15.3
Original code size (Kb)	3.9	1.8	3.5	2.8	3.9
Generated code size (Kb)	25.1	14.3	31.7	18.1	28.6
Size ratio (generated/original)	6.4	7.9	9.1	6.5	7.3

Table 5.1: Domain and compilation statistics.

Set representation, bit vectors vs ordered lists

	Sund		J&L		Str		Deb		Dep	
Program	bits	lists	bits	lists	bits	lists	bits	lists	bits	lists
boyer	2.93	2.94	2.77	8.94	5.07	4.16	1.63	1.45	2.26	2.34
browse	30.86	377.99	55.19	807.95	4.24	4.82	2.20	2.05	1.81	1.91
chatparser	–	–	–	–	19.55	20.60	13.83	12.45	57.92	52.30
crypt	0.83	0.79	0.89	2.23	2.09	2.21	1.17	1.22	1.37	1.25
divide	0.26	0.28	0.16	0.19	0.69	0.73	0.26	0.24	0.56	0.57
fastmu	0.45	0.81	0.30	0.30	1.27	1.32	0.75	1.24	2.30	2.26
flatten	3.37	7.99	18.95	272.16	2.00	2.14	1.34	1.27	4.00	4.40
metaqsort	0.86	1.43	1.08	6.39	0.90	0.88	1.33	0.95	0.84	0.86
poly	4.93	17.33	17.51	326.44	1.61	1.47	1.29	1.01	2.88	2.91
qsort	0.10	0.13	0.03	0.06	0.62	0.66	0.21	0.17	0.56	0.58
queens	0.14	0.16	0.05	0.08	0.55	0.58	0.26	0.22	0.46	0.43
reducer	–	–	–	–	4.12	4.31	3.08	2.58	5.71	6.41
serialise	0.63	0.81	2.48	31.19	1.09	1.07	0.45	0.37	0.79	0.83
analyzer	–	–	–	–	8.27	8.17	4.83	3.90	14.32	12.37
tak	0.06	0.08	0.02	0.04	0.31	0.33	0.08	0.10	0.18	0.19
zebra	–	–	–	–	2.75	5.24	1.62	2.53	8.21	7.53

Table 5.2: Analysis execution times.

and the code to interface the domain with the provided framework. The size of the uncommented domain specification is often close to the size of the published specification of the abstract domain.

The size of the generated code is 6–9 times larger than the specification, for our examples.

The compilation time is important as it determines the turn-around time for the system. A compilation typically takes between 5–20 seconds. The greatest part of this time is spent in the type-checker.

The details of these domain and compilation statistics are given in Table 5.1.

Set representations

As mentioned previously, bit vectors are used to represent sets wherever possible. As can be seen in Tables 5.2–5.3, significant gains can be achieved by using the bit vector representation when the domains rely heavily on union, intersection, and member operations performed on very large sets (`Sund` and `J&L`).

We also tried a different approach to set representation. Instead of explicitly representing every element of a set (often very large power sets), an implicit representation, power set expressions [3], were tried. All sets can be represented with this method, but very compact representation of large sets is only achieved when the contents of the sets are easily described by certain power set formula. Tests showed that this was not the case with our test domains. The large sets in our

Set representation, bit vectors vs ordered lists

Program	Sund	J&L	Str	Deb	Dep
boyer	1.00	3.23	0.82	0.89	1.04
browse	12.25	14.64	1.14	0.93	1.06
chatparser	–	–	0.67	0.90	0.90
crypt	0.95	2.51	1.06	1.04	0.91
divide	1.08	1.19	1.06	0.92	1.02
fastmu	1.80	1.00	1.04	1.65	0.98
flatten	2.37	14.36	1.07	0.95	1.10
metaqsort	1.66	5.92	0.98	0.71	1.02
poly	3.52	18.63	0.91	0.78	1.01
qsort	1.30	2.00	1.06	0.81	1.04
queens	1.14	1.60	1.05	0.85	0.93
reducer	–	–	1.05	0.84	1.12
serialise	1.29	12.58	0.98	0.82	1.05
analyzer	–	–	0.99	0.81	0.86
tak	1.33	2.00	1.06	1.25	1.06
zebra	–	–	1.90	1.56	0.92
Geometric mean	1.74	4.02	1.03	0.95	1.00

Table 5.3: Execution time ratios (ordered lists/bit vectors).

abstract domains tend to be constructed in an incremental fashion that proved to be unsuitable for implicit description.

Efficiency of generated code

Comparisons of hand-coded with auto-generated domains were performed as follows. The sharing analysis of &-Prolog[1] [17] (called `share`, based on Jacob's and Langen's sharing domain [19]) was compared with `J&L`. The hand-coded freeness, sharing, and linearity analysis of &-Prolog (called `shfrson`) was compared with `Sund`. The type, mode, aliasing, linearity, locality and determinism analysis of Reform Prolog[2] [4, 21] was compared with `Dep`. The compared systems were all executed by SICStus Prolog 2.1.9. The compared domains are not identical but similar enough to serve for our approximate comparisons. Only the execution time for analysis is included in the measurements. Program loading, code preparation, presentation of the results and similar phases are left out.

Some entries are left blank in the evaluation. These are for benchmark programs with large numbers of variables and domains that are exponential in the number of variables (`Sund`, `J&L`, and &-Prolog's `share` and `shfrson` domains). The great

[1] Version V0.2.1/C1.1
[2] Version 0.9

Comparison of hand-coded domains and generated domains

Program	share	J&L	ratio	shfrson	Sund	ratio	Reform	Dep	ratio
boyer	3.33	2.77	0.83	3.44	2.93	0.85	4.49	2.26	0.50
browse	9.06	55.19	6.09	6.71	30.86	4.60	1.06	1.81	1.71
chatparser	–	–	–	–	–	–	23.10	57.92	2.51
crypt	0.29	0.89	3.06	0.55	0.83	1.51	0.39	1.37	3.51
divide	0.13	0.16	1.23	0.20	0.26	1.30	0.13	0.56	4.31
fastmu	0.46	0.30	0.65	0.66	0.45	0.68	1.00	2.30	2.30
flatten	33.18	18.95	0.57	14.73	3.37	0.23	2.09	4.00	1.91
metaqsort	0.95	1.08	1.13	3.03	0.86	0.28	0.86	0.84	0.98
poly	0.33	17.51	53.06	0.61	4.93	8.08	0.43	2.88	6.70
qsort	0.06	0.03	0.50	0.11	0.10	0.91	0.21	0.56	2.67
queens	0.10	0.05	0.50	0.14	0.14	1.00	0.21	0.46	2.19
reducer	–	–	–	–	–	–	3.85	5.71	1.48
serialise	1.92	2.48	1.29	0.51	0.63	1.24	0.76	0.79	1.04
analyzer	–	–	–	–	–	–	9.03	14.32	1.59
tak	0.05	0.02	0.40	0.09	0.06	0.67	0.06	0.18	3.00
zebra	–	–	–	–	–	–	0.52	8.21	15.79
Geometric mean			1.39			1.06			2.30

Table 5.4: Execution time ratios (generated/hand-coded).

fluctuations in the execution time of some of the benchmarks are due to the exponential behaviour of the above mentioned domains. A small change in the number of variables in a clause, due to differences of program normalization, can result in significant time differences.

The efficiency of the generated code compares well with similar hand-coded implementations. As can be seen in Table 5.4, the performance of the generated code is, on average, well within an order of magnitude of the hand-coded domains. Most of the time the performance of the generated code is within a factor 0.5–3 of the hand-coded domain.

Related work

The most well-known predecessors to Igor are the tools Yacc [18], a parser generator, and Lex [20], a tool that generates lexical analyzers. These were developed at Bell Laboratories in the mid-seventies and are still two of the most widely used tools for compiler development.

Venkatesh [30] designed a denotational semantics specification language augmented with a collecting semantics mechanism for program analysis. The abstract domains, that are similar to Igor's, are combined with semantic functions over these domains to make up an analyzer. The specifications are interpreted rather than compiled.

The Z1 system [32] allows the programmer to specify an interprocedural analyzer, consisting of an abstract interpreter and an abstract domain, which is compiled into executable C-code. Our system extends the capabilities of Z1 with disjunctive and structure-based domains and more flexible projection operations. Igor is furthermore substantially faster. We do not, however, include the analysis framework in the specifications.

Tjiang [28] describes a tool that greatly simplifies the implementation of optimizers by using high-level specifications to combine several simpler optimization specifications. This tool works with flow-graphs, using a data-flow analysis method called *path simplification*, and is aimed at imperative rather than declarative languages.

Genesis [31] is a tool that generates complex optimizers from specifications that describes the combination of several simpler optimizations. The tool is primarily intended to help prototyping optimizers during the design of compilers for parallel imperative languages.

Cousot and Cousot [9] proposed theoretical methods for systematic design of new abstract domains out of old ones, by combinations of domains and application of various transformations on domains. Related methods have been developed by Cortesi et al [7] who propose two kinds of support for domain construction. *Generic pattern domains* is software support for upgrading simpler domains to include structural information. This upgrade results in more accurate domains.

Open products is a method for combining domains to obtain a more sophisticated domain. The implementation of new abstract domains by combining old ones is also described in [6]. This method can successfully be used in IGOR-specifications.

Van Roy [26] notes that the actual use of the analysis results is less well-researched, as compared to design of generic analysis frameworks or abstract domains. Getzinger [15] performs an evaluation of the advantages gained from of a large number of domains, when used to compile logic programs. In IGOR, this type of domain evaluation is facilitated by the provided support for specifying code annotations, information which can be utilized by any subsequent compilation phase.

Conclusion

7.1 CONCLUSION

We have developed and implemented methods for automatically generating often used abstract domain operations. The generated code has been shown to be sufficiently efficient for domain prototyping and comparable with hand-coded domains.

We envision language implementors and researchers to use the Igor tool for several purposes:

- To reduce the effort required for producing a compiler that uses static analysis.
- To reduce the effort required for quantitative evaluation of new domain designs.
- To test and debug ideas and specifications during domain design.

The tool should be instrumental in helping to change the task of implementing static analysis domains from being a black art into a routine task on the same level as using Lex or Yacc for lexical analysis and parsing.

7.2 DISCUSSION AND FUTURE WORK

The language does not provide support for type graphs [16]. Language support for handling graphs should be added. One possibility would be *graph explorations* [14], a language construct similar to set expressions and list comprehensions, but intended to express graph algorithms. Definite boolean functions [2] is another representation used by abstract domain designers that need some basic support in the language to be easily used in Igor-specifications.

The disjunctive lattice type automatically augments a lattice with disjunctive elements. It would be possible to automatically augment lattices with other types of elements, i.e. complementary elements, using similar methods.

Among the most complicated domains to implement are domains that contain information on the form or structure of data. The generic pattern domains described by Cortesi et al [7] alleviate much of the design work on these domains by automatically extending non-structural domains with structure information. It would be possible and desirable to add such a method to IGOR.

IGOR can be told to annotate an analyzed program with the result of the analysis. The annotation is made with complete or partial domain elements and is local to clauses. A useful extension would be to allow global, arbitrarily computed annotations. This would make it possible to calculate domain or program statistics from the analysis result.

The current type system should be replaced with a stronger type system that is able to handle type inclusions. One such type system is Aiken's set constraint based type system [1]. This would enable IGOR to detect more situations where data representation optimizations can be applied and the more precise types would facilitate the discovery of more specification errors.

The generated code is quite efficient but there is room for improvement. High-level optimizations such as partial evaluation, common subexpression elimination and code motion would all contribute to closing the gap between hand-coded and computer generated code. These optimizations would profit from the information supplied by the stronger type system.

IGOR currently limits the user to specifying only the abstract domain component of an analyzer. The framework must be written by the designer. An expansion of the language could be made to enable the user to also specify the abstract interpretation framework of the analyzer and have it automatically generated. This modification would simplify the development of abstract interpreters for non-standard logic programming semantics or for different languages.

The semantics of the domain operations are only empirically tested. Correctness proofs for these operations are desirable.

Acknowledgments

Thanks to Håkan Millroth, my thesis advisor, who has steered me throughout the long journey towards the end and to Thomas Lindgren and Per Mildner whose continuous harassments have improved the work considerably. I would also like to thank all the other members of the Computing Science Department of Uppsala University who have contributed with helpful ideas and suggestions.

The Analyzer

This appendix describes the current Prolog program analysis framework included in the system. It is possible to interface your own analyzer with the generated domain code. A pseudo-code version of the current analyzer framework is given below to facilitate the understanding of the relationship between the analyzer and the generated domain code.

A.1 ANALYZER FRAMEWORK

The analysis consists of the following phases:

1. **Consulting.** Load the program to be analyzed.
2. **Code preparation.** Transform all disjunctions, negations and if-then-else-expressions into Horn clauses by creating new clauses. Normalize all clauses according to the current normalization-flag (see below). Harmonize all clause heads within a predicate to use the same head variables.
3. **Compute call graph.** Compute the call graph of the transformed program. Compute the strongly connected components of this graph. Topologically sort the strongly connected components. This information is used by the analyzer to keep an priority queue of goals to be analyzed. The fewer predicates a predicate is dependent upon, the higher priority it will be given in the queue. This minimizes the reanalysis of predicates when the analysis result of a predicate they are dependent upon changes.
4. **Initialize analyzer.** Set initial entry and exit descriptions.
5. **Analyze program.** Perform the analysis of the program, and, if not running quietly, report the result.

The analyzer framework is based on Getzinger's generic abstract interpretation framework [15]. The global data used in the analysis algorithm, as shown in the figure, are:

The analyzer framework

```
procedure analyze()
    if Predicates ≠ ∅ then
        Pred := choose(Predicates)
        analyze_disj(Pred)
        analyze()

procedure analyze_disj(Pred)
    TailDesc := ⊥
    for each Clause ∈ clauses(Pred) do
        ExitDesc := analyze_conj(Clause, entry_desc(Pred))
        TailDesc := TailDesc ⊔ ExitDesc
    if not TailDesc ⊑ exit_desc(Pred) then
        exit_desc(Pred) := TailDesc ⊔ exit_desc(Pred)
        Predicates := Predicates ∪ dependents(Pred)

procedure analyze_conj(Clause, HeadDesc)
    Desc := clause_entry(Clause, HeadDesc)
    for each Goal ∈ body(Clause) do
        Desc := analyze_goal(Goal, Desc)
        exit loop if Desc = ⊥
    return clause_exit(Clause, Desc)

procedure analyze_goal(Goal, CallDesc)
    if is_builtin_pred(Goal) then
        return builtin_pred(Goal, CallDesc)
    else
        Pred := get_matching_pred(Goal)
        HeadDesc := pred_entry(Goal, head(Pred), CallDesc)
        if not reached(Pred) or not HeadDesc ⊑ entry_desc(Pred) then
            entry_desc(Pred) := entry_desc(Pred) ⊔ HeadDesc
            reached(Pred) := true
            analyze_disj(Pred)
        return pred_exit(head(Pred), Goal, CallDesc, exit_desc(Pred))
```

The analyzer framework is based on Getzinger's generic framework.

Predicates This is a priority queue of predicates to be analyzed based on the topologically sorted strongly connected component of the call graph of the program being analyzed. It is initialized with the top-predicate `top/0`.

entry_desc(Pred) This is a table containing the entry descriptions of all predicates. All entries are initially set to $\bot$.

exit_desc(Pred) This is a table containing the exit descriptions of all predicates. All entries are initially set to $\bot$.

dependents(Pred) This table contains all dependencies between predicates in the program being analyzed. If the description of a predicate changes during analysis, all predicates that are dependent on, i.e., calls, this predicate must be re-analyzed.

reached(Pred) This boolean valued table indicates whether a predicate has been reached, i.e., exposed to at least one pass of analysis, or not, during the analysis. Initially no predicate has been reached.

The flags controlling the analyzer are:

- `normalize([off|head|nonground|all])` Decides which type of normalization to use on the program to be analyzed. We will use the example clause

  ```
  p(1,A,B,h(B,C)) :- q(f(A,B),[a],g(A,A),C).
  ```

  to illustrate the types of normalization.

  - `off`: No normalization is applied and no harmonization of clause heads is performed. This type of normalization cannot used by the current analyzer. The example clause would be left as is.
  - `head`: The head arguments must be variables. This set of variables must be linear, i.e., contain no repeated occurrence of any variable. This results in:

    ```
    p(H1,A,B,H2) :- H1=1, H2=h(B,C),
                    q(f(A,B),[a],g(A,A),C).
    ```

    and the harmonization of all heads of predicate `p/4` to contain the variables `H1`, `A`, `B`, and `H2` in that order.
  - `nonground`: In the addition to the normalization of the heads, all unifications must be on the form *Variable*=*Variable*, *Variable*=*GroundTerm* or *Variable*=f(*Variable*$_1$, ..., *Variable*$_n$) and all goals must be on the form

$p(Variable_1, ..., Variable_n)$. The sets $\{Variable_1, ..., Variable_n\}$ must be linear. This results in:

```
p(H1,A,B,H2) :- H1=1, H2=h(B,C), V1=f(A,B),
                V2=[a], V3=A, V4=g(A,V3),
                q(V1,V2,V4,C).
```

- `all`: In addition to the normalization described above, all ground terms are normalized, i.e., the unification form *Variable=GroundTerm* is replaced by *Variable=Atom*. This results in:

```
p(H1,A,B,H2) :- H1=1, H2=h(B,C), V1=f(A,B), V2=a,
                V3=[], V4=[V2|V3], V5=A, V6=g(A,V5),
                q(V1,V4,V6,C).
```

More normalization results in simpler goals and unifications, but also introduces more unifications and variables to the program. Thus more normalization simplifies the definition of domain functions but increases the size of descriptors and the number of goals to be analyzed. As many domains have a computational complexity with the number of variables in a descriptor as the most significant parameter, an increased number of variables may degrade the speed of the analysis. An example of the effects of normalization on some benchmarks programs is provided in Appendix B.

The default setting of this flag is `normalize(all)`. It is always wise to set this flag with a command in the domain specification file, as it can be quite costly to use a higher degree of normalization than the domain was designed for. A lower degree of normalization, than domain was designed for, will probably not work at all.

- `terminate_on_bot([on|off])` Decides if the analyzer should terminate the analysis of a clause body when the descriptor equals the bottom ($\perp$) descriptor. The default is `on`.

- `statistics([on|off])` Turns on or off some statistics about the program being analyzed. The default is `off`.

- `annotation(AnnSpec)` This flag controls the annotation of the program. `AnnSpec` is an annotation specification. See section A.3.

A.2 ANALYZER INTERFACE

The following predicates are used by the analyzer and should be defined in the domain specification.

- `pred_entry(Goal, Head, CallDesc, EntryDesc)` This predicate is called when a call to a goal enters a predicate. `Goal` is the goal that initiated the entry, `CallDesc` is the descriptor of the environment of this goal, `Head` is the

head of the predicate about to be entered, and the return value, `EntryDesc`, contains the resulting description of the initial environment of the predicate head.

- `pred_exit(Head, Goal, CallDesc, ExitDesc, SuccDesc)` This predicate is called when all clauses of a predicate has been analyzed and the resulting information is to be transfered from the environment of the predicate to the environment of the invoking goal. `Goal` is the goal that initiated the entry of the predicate, `CallDesc` is the descriptor of the environment of this goal, `Head` is the head of the predicate entered, `ExitDesc` is the description of the predicate head after analysis, and, the return value, `SuccDesc`, is the descriptor of the goal's environment after the call is completed.

- `clause_entry(Clause, HeadDesc, EntryDesc)` This predicate is called when a clause is to be entered. `Clause` is the clause about to be entered, `HeadDesc` is the descriptor of the environment of the clause head, and the return value `EntryDesc` is the descriptor of the initial environment of this clause.

- `clause_exit(Clause, ExitDesc, TailDesc)` This predicate is called when the analysis of a clause is finished. `Clause` is the clause having been analyzed, `ExitDesc` is the descriptor of the final environment of the class and the return value `TailDesc` is the environment of the clause head after analysis of the clause body.

The following predicates are generated from the type definitions and the `builtin/2-3` definitions of the domain specification.

- `desc_bot(Head, Bot)` returns the bottom descriptor, `Bot`, based on a predicate head `Head`.

- `desc_leq(D1, D2)` is true if `D1` $\sqsubseteq$ `D2`.

- `desc_join(D1, D2, D)` returns `D1` $\sqcup$ `D2` in `D`.

- `is_builtin_pred(Goal)` is true if `Goal` calls a built-in predicate.

- `builtin_pred(Goal, CallDesc, SuccDesc)` This predicate is called when a call to a built-in predicate is to be analyzed. `Goal` is the goal calling the predicate, `CallDesc` is the descriptor of the environment of the goal, and the return value `SuccDesc` is the descriptor of the environment of the goal after abstract execution of the built-in predicate.

The use of these predicates in the analyzer results in type relationships that the type checker cannot access, as it only infers types of the domain specification.

To remedy this, a special file, `interface_types.ad`, contains a dummy definition which reflects the type relationships caused by the use of the interface predicates by the analyzer. This dummy function must be provided by the designer wishing to interface his own analyzer with generated domain code.

A.3 COMMANDS

The following commands are available to control the analyzer:

- `analyze(`*Filename*`)` Analyzes the Prolog program contained in *Filename*. If no file-name suffix is given ".pl" is assumed. The file must contain one occurrence of the `top/0` predicate. This predicate decides where to start the analysis. The body of the `top/0` predicate should be a query that reflects the use of the program to be analyzed.

- `annotate(`*Filename*`)` Analyzes the Prolog program contained in *Filename*, annotates (see the manual [24]) the program with the result, and writes the annotated program to the file *Filename*`.an`.

- `debug_level(`*Level*`)` sets the debugging level to *Level*. The specification must have been compiled with `cons/1` to be debuggable. More information on debugging levels can be found in the IGOR manual [24].

- `timing_level(`*Level*`)` set the timing level to *Level*. More information on timing levels can be found in the IGOR manual [24].

Benchmarks

	Original		Normalized		
Program	Max	Total	Max	Total	Description
boyer	7	352	47	937	a Boyer-Moore theorem prover
browse	12	145	159	375	a pattern matching benchmark
chat_parser	3	1646	42	3274	a natural language parser
crypt	19	64	49	167	a crypto cracker
divide	5	38	21	90	symbolic derivation
fast_mu	13	107	22	213	solves a theorem in Hofstader's mu-math system
flatten	13	233	19	362	flatten Prolog into a simpler syntax
meta_qsort	7	81	105	239	a Prolog meta-interpreter
poly	8	106	27	223	symbolic polynomial expansion
qsort	7	20	9	36	quicksort
queens	5	30	10	53	solves the eight queens problem
reducer	17	494	26	917	a graph reducer for T-combinators
serialise	7	44	52	124	compute unique serial numbers
simple_analyzer	8	693	185	1111	a simple mode analyzer
tak	10	15	16	25	a highly recursive integer function (Takeuchi)
zebra	78	127	131	204	a puzzle solver

Table B.1: The benchmarks used in the evaluation

The first two columns shows the maximum number of variables per clause and the total number of variabels in the the original benchmark program. The next two columns show the same data after full normalization (see page 42) of the program. The number of variables in a clause is one of the most important factors determining the efficiency of many abstract domains.

Examples

This appendix provides examples of some domain specifications. For ease of reading each example is divided into the following sections: type definitions, analyzer interface, abstract unification, and analysis of built-in predicates. The last section is often omitted to save space.

C.1 DEFINITE GROUNDNESS, DEFGR.AD

This is a very simple definite groundness domain. There is no abstract unification section in this specification since this function is provided directly by the interface definitions.

Type definitions

```
type desc(Clause) => invset(variables(Clause)).
```

Analyzer interface and abstract unification

```
pred_entry(Goal, Head, InitDesc) =>
  {arg(I,Head) | I <- (1 .. arity(Goal)),
                 is_subset(vars(arg(I,Goal)), InitDesc)}.

clause_entry(_Clause, HeadDesc) => HeadDesc.

clause_exit(Clause, LastDesc) => LastDesc * vars(head(Clause)).

pred_exit(Head, Goal, InitDesc, TailDesc) =>
  InitDesc + union({vars(arg(I,Goal)) | I <- (1 .. arity(Goal)),
                                is_member(arg(I,Head), TailDesc)}).
```

Analysis of built-in predicates

```
builtin(X = Y, D) =>
  ( is_constant(Y) \/ is_subset(vars(Y), D) -> {X} + D
  ; is_member(X, D) -> D + vars(Y)
  ; D ).
builtin(X is Y, D) => ground(Y, ground(X, D)).
builtin({<,>,=<,>=,=\=,=:=}, op(X,Y), D) => ground(X, ground(Y, D)).
builtin({ground,number}, op(X), D) => ground(X, D).
builtin({==,\==,@<,@>,@=<,@>=,sort,keysort}, op(_,_), D) => D.
builtin({!,nl,true,fail}, op, D) => D.
builtin(functor(_T,F,A), D) => ground(F, ground(A, D)).
builtin(arg(N,_T,_A), D) => ground(N, D).
builtin(_ =.. _, D) => D.
builtin({var,write,nonvar}, op(_), D) => D.
builtin({atom, atomic, integer, number}, op(X), D) => ground(X, D).
builtin({name}, op(A,B), D) => ground(A, ground(B, D)).
builtin(compare(Op, _, _), D) => ground(Op, D).

ground(V, D) => {V} + D.
```

C.2 A SIMPLE TYPE DOMAIN, DEB.AD

This is one of Debray's substitution closed type domains [12]. It keeps track on integers, lists of integers, constants, lists of constants, lists, and non-variables.

```
%                               any
%                                |
%                                nv
%                                |\
%                              list c
%                                |/ \
%                             clist  integer
%                                \  /
%                                 \/
%                                 none
:- set_flag(normalize(all)).
```

Type definitions

```
type subst => lattice([[any,nv,list,clist,none],
                       [c,clist],[nv,c,integer,none]]).

type desc(Head) => variables(Head) -> subst.
```

Analyzer interface

```
pred_entry(Goal, Head, InitDesc) =>
  restrict_to_vars(vars(Head),
                   head_amgu(rename(args_of(Goal)),
                             args_of(Head),
                             HeadDesc + rename(InitDesc)))
  where
  HeadDesc = {(V,any) | V <- vars(Head)}.

clause_entry(Clause, HeadDesc) =>
  HeadDesc + {(V,any) | V <- vars(Clause), not(is_member(V, HV))}
  where
  HV = vars(head(Clause)).

clause_exit(Clause, LastDesc) =>
  restrict_to_vars(vars(head(Clause)), LastDesc).

pred_exit(Head, Goal, InitDesc, TailDesc) =>
```

```
  restrict_to_vars(DomInitDesc, head_amgu(args_of(Goal),
                                          rename(args_of(Head)),
                                          InitDesc + rename(TailDesc)))
  where DomInitDesc = {V | (V,_) <- InitDesc}.

restrict_to_vars(Vars, Desc) =>
  {(V,X) | (V,X) <- Desc, is_member(V, Vars)}.
```

Abstract unification

```
head_amgu([], [], D) => D.
head_amgu([A|As], [B|Bs], D) => head_amgu(As, Bs, amgu(A, B, D)).

amgu(A, B, D) =>
  ( is_var(B) -> (update(B, Inst, update(A, Inst, D))
                  where Inst = subst_meet(D @ A, D @ B))
  ; update_vars(vars(B), Inst, update(A, Inst, D))
    where
    Inst = subst_meet(D @ A, term_inst(B, D))).

term_inst(T, D) =>
  ( is_cons(T) -> ( forall(V <- vars(T), is_desc_ground(V, D)) -> clist
                  ; list )
  ; is_integer(T) -> integer
  ; is_constant(T) -> c
  ; forall(V <- vars(T), is_desc_ground(V, D)) -> c
  ; nv ).

update_vars(S, ParentInst, D) =>
  ( S == {} -> D
  ; (update_vars(Vs, ParentInst,
                 update(V, subst_meet(D @ V, child_inst(ParentInst)), D))
     where (V,Vs) = set_first_rest(S))).

child_inst(none) =>  none.
child_inst(integer) => none.
child_inst(c) => c.
child_inst(clist) => c.
child_inst(list) => any.
child_inst(nv) => any.
child_inst(any) =>  any.
```

C.3 JACOBS'S AND LANGEN'S SHARING DOMAIN, JL.AD

This is Jacobs's and Langen's simple sharing domain [19]. It has no knowledge of linearity and can be extremely time consuming when analyzing clauses containing many variables.

```
:- set_flag(normalize(head)), set_flag(terminate_on_bot(off)).
```

Type definitions

```
type desc(Term) => set(set(variables(Term))).
```

Analyzer interface

```
pred_entry(Goal, Head, InitDesc) =>
  restrict_to_vars(vars(Head),
     amgu(rename(Goal), Head, HeadDesc + rename_desc(InitDesc)))
  where
  HeadDesc = {{V} | V <- vars(Head)}.

clause_entry(Clause, HeadDesc) =>
  HeadDesc + {{V} | V <- vars(Clause), not(is_member(V, HV))}
  where
  HV = vars(head(Clause)).

clause_exit(Clause, LastDesc) =>
  restrict_to_vars(vars(head(Clause)), LastDesc).

pred_exit(Head, Goal, InitDesc, TailDesc) =>
  restrict_to_vars(union(InitDesc),
                   amgu(Goal,
                        rename(Head),
                        InitDesc + rename_desc(TailDesc))).

restrict_to_vars(Vars, Desc) =>
  {{X | X <- S, is_member(X, Vars)} | S <- Desc} \ {{}}.

rename_desc(Desc) => {{rename(X) | X <- S} | S <- Desc}.
```

Abstract unification

```
amgu(A, B, D) => amgu_args(unify(A, B), D).

amgu1(At, Bt, D) =>
 D \ (A + B) + {X + Y | X <- CA, Y <- CB}
 where
 A = rel(At, D), B = rel(Bt, D),
 CA = closure(+, 2, A), CB = closure(+, 2, B).

amgu_args([],D) => D.
amgu_args([(A,B)|As],D) => amgu_args(As, amgu1(A, B, D)).

rel(T, D) =>
  {X | X <- D, VT * X \= {}} where VT = vars(T).
```

C.4 SUNDARARAJAN'S DOMAIN, SUND.AD

Sundararajan's domain [27] keeps track of definite freeness, linearity, and aliasing. This domain is more precise and faster than Jacobs's and Langen's domain, but can still be extremely time consuming when analyzing clauses with a large number of variables. This domain includes an example of inclusion of Prolog predicates into the specification.

```
:- set_flag(normalize(head)), set_flag(terminate_on_bot(off)).
```

Type definitions

```
type free(Clause) => invset(variables(Clause)).

type repeat(Clause) => set(variables(Clause)).

type sharing(Clause) => set(set(variables(Clause))).

type desc(C) => (free(C), repeat(C), sharing(C)).
```

Analyzer interface

```
pred_entry(Goal, Head, InitDesc) =>
  restrict_to_vars(vars(Head), aunify(Goal, InitDesc, Head, HeadDesc))
  where
  HV = vars(Head),
  HeadDesc = (HV, repeat_bot(Head), {{X} | X <- HV}).

clause_entry(Clause, (FH, RH, SH)) =>
  (FH + CV, RH, SH + {{V} | V <- CV})
  where
  HV = vars(head(Clause)),
  CV = {V | V <- vars(Clause), not(is_member(V, HV))}.

clause_exit(Clause, LastDesc) =>
  restrict_to_vars(vars(head(Clause)), LastDesc).

pred_exit(Head, Goal, (FG, RG, SG), TailDesc) =>
    restrict_to_vars(union(SG),
                     aunify(Head, TailDesc, Goal, (FG, RG, SG))).

aunify(Goal, (F1, R1, S1), Head, (F2, R2, S2)) =>
  amgu(rename(Goal), Head, (F2 + FR, R2 + RR, S2 + SR))
  where
```

```
  (FR, RR, SR) = (rename_set(F1),
                  rename_set(R1),
                  {rename_set(S) | S <- S1}).

restrict_to_vars(Vs, (F,R,S)) =>
  (rtvs(Vs, F), rtvs(Vs, R), {rtvs(Vs, X) | X <- S} \ {{}}).

rtvs(Vs, S) => {X | X <- S, is_member(X, Vs)}.

rename_set(S) => {rename(X) | X <- S}.
```

Abstract unification

```
amgu(Goal, Head, (Free, Repeat, Sharing)) =>
  propagate_frs((Free0, Repeat0, Sharing0), Theta0)
  where
    Theta = unify(Goal, Head),
    G0 = vars(Goal) + vars(Head) \ union(Sharing),
    G1 = G0 + {X | (V,T) <- Theta, X <- vars(T), is_member(V,G0)},
    G = G1 + {V | (V,T) <- Theta, is_subset(vars(T), G1)},
    Theta0 = {(X, update(T, G)) | (X,T) <- Theta,
                                  not(is_member(X, G))},
    Sharing0 = Sharing \ srelevant(G, Sharing),
    Repeat0 = Repeat \ G,
    Free0 = Free \ union(srelevant(G, Sharing)).

multioccurs(T, S) =>
  #MT \= #VT \/
  exists(X <- VT, Y <- VT,
         X \= Y /\ exists(S1 <- S, is_subset({X}+{Y}, S1)))
  where
  MT = multivars(T),
  VT = set(MT).

propagate_frs(ASub, Sigma) =>
  (Sigma == {} -> ASub
  ; propagate_frs(propagate_one_binding(ASub, set_first(Sigma)),
                                              set_rest(Sigma))).

propagate_one_binding((F,R,S), (V,T)) =>
  (F2,R3,S1)
  where
  A = srelevant({V}, S),
  B = trelevant(T, S),
  B1 = (is_member(V, R) -> closure(+,2, B) ; B),
  A1 = ((vars(T) * R \= {} \/ multioccurs(T,S)) -> closure(+,2, A) ; A),
```

```
  S1 = (S \ (A + B)) + {Ai + Bi | Ai <- A, Bi <- B},
  R1 = R + (is_member(V,R) -> union(B) ; {}),
  R2 = R1 + ((vars(T) * R \= {} \/ multioccurs(T,S1)) -> union(A) ; {}),
  R3 = R2 + union({X * Y | X <- A1, Y <- B1}),
  F1 = F \ (not(is_member(V,F)) -> union(trelevant(T,S1)) ; {}),
  F2 = F1 \ (is_nonvar(T) \/ (is_var(T) /\ not(is_member(T,F1)))
             -> union(trelevant(V,S)) ; {}).

srelevant(T, S) => {X | X <- S, T * X \= {}}.
trelevant(T, S) => srelevant(vars(T), S).

prolog term:update(term:T, set(term):G).
update(T,G,GT) :- is_var(T),!,(is_member(T,G) -> GT = ground ; GT = T).
update(T,G,GT) :- T=..[F|Args],update_args(Args,G,GArgs),GT=..[F|GArgs].
update_args([],_,[]).
update_args([T|Ts],G,[GT|GTs]) :- update(T,G,GT),update_args(Ts,G,GTs).
```

C.5 A STRUCTURE DOMAIN, STR.AD

This is an example of a domain using an attributed recursive disjunctive lattice. It keeps track of the structure and groundness of terms using the notation `(str((Functor,Arity), [TypeOfArg1,...,TypeOfArgN]), Groundness)`. It uses a projection to keep the domain elements within a certain depth and width.

```
:- set_flag(normalize(all)).
```

Type definitions

```
type ground => lattice([[any,gnd,none]]).

type str => disj_lattice([[any,str,str(F,list(type)),none]]).

type type => (str, ground).

type desc(C) => variables(C) -> type.

desc_proj(D) => {(V,depth_k(T,3,3)) | (V,T) <- D}.
```

Analyzer interface

```
pred_entry(Goal, Head, InitDesc) =>
  restrict_to_vars(vars(Head), head_amgu(rename(args_of(Goal)),
                                         args_of(Head),
                                         HeadDesc + rename(InitDesc)))
  where
  HeadDesc = {(V,type_top) | V <- vars(Head)}.

clause_entry(Clause, HeadDesc) =>
  HeadDesc + {(V,type_top) | V <- vars(Clause), not(is_member(V, HV))}
  where
  HV = vars(head(Clause)).

clause_exit(Clause, LastDesc) =>
  restrict_to_vars(vars(head(Clause)), LastDesc).

pred_exit(Head, Goal, InitDesc, TailDesc) =>
  restrict_to_vars(DomInitDesc, head_amgu(args_of(Goal),
                                          rename(args_of(Head)),
                                          InitDesc + rename(TailDesc)))
  where DomInitDesc = {V | (V,_) <- InitDesc}.
```

```
restrict_to_vars(Vars, Desc) =>
  {(V,X) | (V,X) <- Desc, is_member(V, Vars)}.
```

Abstract unifications

```
head_amgu([], [], D) => D.
head_amgu([A|As], [B|Bs], D) => head_amgu(As, Bs, aunify(A, B, D)).

aunify(A,B,D) =>
  update(A, Meet, D1)
  where
  Meet = type_meet(D @ A, abs_term(B, D)),
  D1 = ( is_var(B) -> update(B, Meet, D)
       ; is_constant(B) -> D
       ; subtype_vars(args_of(B), 1, Meet, D) ).

subtype_vars([], _, _, D) => D.
subtype_vars([V|Vs], N, (S,G), D) =>
    ( S == str(_, Args) -> update(V, type_meet(D1 @ V, nth(N, Args)), D1)
    ; update(V, type_meet((any,G),D1 @ V), D1) ) % Fel?
    where
    D1 = subtype_vars(Vs, N+1, (S,G), D).

abs_term(T, D) =>
  ( is_var(T) -> D @ T
  ; is_constant(T) -> (str(functor_arity_of(T),[]),gnd)
  ; ( (str(functor_arity_of(T),Args),ground_join((G | (_,G) <- Args)))
    where
    Args = (abs_term(A,D) | A <- args_of(T)) ) ).
```

Depth- and width-limiting projection

```
depth_k((S,G), K, W) =>
  ( K =< 0 ->
    ( S == str(_,_) ->
      (str,G)
    ; is_disjunctive(S) ->
      (str_basic_join({str_depth_k(A,0,W) | A <- disjuncts(S)}),G)
    ; (S,G) )
  ; ( S == str(FA,Args) ->
      (str(FA,(depth_k(A,K-1,W) | A <- Args)),G)
    ; is_disjunctive(S) ->
```

```
      (width(make_disjunctive(
             {str_depth_k(A,K,W) | A <- disjuncts(S)}), W),G)
    ; (S,G) ) ).

str_depth_k(S, K, W) =>
  S1 where (S1,_) = depth_k((S,gnd), K, W).

width(D ,W) =>
  ( #disjuncts(D) > W ->
    str_basic_join({str_depth_k(A,0,W) | A <- disjuncts(D)})
  ; D ).
```

Bibliography

1. A. Aiken, E.L. Wimmers & T.K Lakshman, Soft Typing with Conditional Types, *Proc. Symp. Principles of Programming Languages, POPL'94*, 1994.

2. T. Armstrong, K. Marriott, P. Schacte & H. Sondergaard, Boolean functions for dependency analysis: Algebraic properties and efficient representation, *Static Analysis Symp. 94*, Springer LNCS 864, Springer-Verlag, 1994.

3. D. Berque, R. Cecchini, M. Goldberg, R. Rivenburgh, The SetPlayer System for Symbolic Computation on Power Sets, *J. of Symbolic Computation*, Vol. 14, pp. 645–662, 1992.

4. J. Bevemyr, T. Lindgren & H. Millroth, Exploiting recursion-parallelism in Prolog, *Intl. Conf. PARLE-93* (eds. A. Bode, M. Reeve & G. Wolf), Springer LNCS 694, Springer-Verlag, 1993.

5. M. Carlsson, J. Widén, J. Andersson, S. Andersson, K. Boortz, H. Nilsson, T. Sjöland, *SICStus Prolog User's Manual*, Swedish Institute of Computer Science, 1993.

6. M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda & M. Hermenegildo, Improving abstract interpretations by combining domains, *Proc. Symp. Partial Evaluation and Semantics-Based Program Manipulation, PEPM'91*, Yale University, Connecticut, 1991.

7. A. Cortesi, B. Le Charlier & P. van Hentenryck, Conceptual and software support for abstract domain design: Generic structural domain and open product, *Proc. Symp. Principles of Programming Languages, POPL'94*, 1994.

8. P. Cousot & R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in *Conference Record of the 4th ACM Symposium on Principles of Programming Languages*, 1977.

9. P. Cousot & R. Cousot, Systematic design of program analysis frameworks, *Proc. 6th Conf. Principles of Programming Languages*, pp. 269–282, 1979.

10. P. Cousot & R. Cousot, Abstract interpretation and application to logic programs, *J. of Logic Programming*, Vol. 13, pp. 103-179, 1992.

11. S.K. Debray, Static inference of modes and data dependencies in logic programs, *ACM Trans. Programming Languages and Systems*, Vol. 11, No. 3, pp. 418–450, July 1989.

12. S.K. Debray, Efficient dataflow analysis of logic programs, *J. ACM*, Vol. 39, No. 4, October 1992.

13. S.K. Debray, QD-Janus: a sequential implementation of Janus in Prolog, *Software — Practice and Experience*, Vol. 23, No. 12, December 1993.

14. M. Erwig, Graph Algorithms = Iteration + Data Structures?, *Graph-Theoretic Concepts in Computer Science*, LNCS 657, 1992.

15. T.W. Getzinger, *Abstract interpretation for the compile-time optimization of logic programs*, Ph.D. Thesis, University of South California, Report 93/09, 1993.

16. P. van Hentenryck, A. Cortesi & B. Le Charlier, Type analysis of prolog using type graphs, *J. Logic Programming*, Vol. 22, No. 3, 1995.

17. M. Hermenegildo & K. Greene, The &-Prolog system: Exploiting independent and-parallelism, *New Generation Computing*, Vol. 9, pp. 233–257, 1991.

18. S.C. Johnson, Yacc — Yet another compiler-compiler, Comp. Sci. Tech. Rep. No. 32, Bell Laboratories, July 1975.

19. D. Jacobs & A. Langen, Accurate and efficient approximation of variable aliasing in logic programs, *Proc. North American Conf. Logic Programming 1989*, pp. 154–165, 1989.

20. M.E. Lesk, Lex — A lexical analyzer generator, Comp. Sci. Tech. Rep. No. 39, Bell Laboratories, October 1975.

21. T. Lindgren, *The compilation and execution of recursion-parallel Prolog on shared-memory multiprocessors*, Licentiate of Philosophy Thesis, Uppsala Theses in Computer Science 18/93, November 1993.

22. T. Lindgren, personal communication.

23. R. Milner, A theory of type polymorphism in programming, *J. of Computer and System Science*, Vol. 17, pp. 348-375, 1978.

24. M. Nordin, *Manual of* IGOR*: A tool for developing abstract domains for Prolog*, technical report, Computing Science Dept., Uppsala University, forthcoming, 1995.

25. S.L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice Hall, 1987.

26. P.L. van Roy, *Can Logic Programming Execute as Fast as Imperative Programming?*, Ph.D. Thesis, UCB/CSD 90/600, Computer Science Division (EECS), University of California, Berkeley, 1990.

27. R. Sundararajan, An abstract interpretation scheme for groundness, freeness, and sharing analysis of logic programs, Technical Report CIS-TR-91-06, Dept. of Computer and Information Science, University of Oregon, 1991

28. S.W-K. Tjiang, *Automatic Generation of Data-Flow Analyzers: A Tool for Building Optimizers*, Ph.D. Thesis, Dept. of Computer Science, Stanford University, July 1993.

29. D.A. Turner, Recursion equations as a programming language, *Functional Programming and Its Applications*, Darlington *et al.*, editors, Cambridge University Press, 1982.

30. G.A. Venkatesh, A framework for construction and evaluation of high-level specifications for program analysis techniques, *SIGPLAN Conf. Programming Language Design and Implementation, PLDI'89*, pp. 1–12, 1989.

31. D. Whitfield & M.L. Soffa, The Design and Implementation of Genesis, *Software — Practice and Experience*, Vol. 24, No. 3, pp. 307–325, 1994.

32. K. Yi & W.L. Harrison III, Automatic generation and management of interprocedural program analyses, *The 20th Annual ACM Symposium on Principles of Programming Languages*, January 1993.

UPMAIL
Computing Science Department
Uppsala University
Box 311
S-751 05 UPPSALA
Sweden

Phone: Nat 018 – 18 25 00
Int +46 18 18 25 00

Fax: Nat 018 – 52 12 70
Int +46 18 52 12 70

ISSN 0283-359X