PROPERTY INFERENCE FOR MAPLE: AN APPLICATION OF ABSTRACT INTERPRETATION

By
STEPHEN A. FORREST, B.MATH.

A Thesis
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements for the Degree of

Master of Science
Department of Computing and Software
McMaster University

MASTER OF SCIENCE(2007)
(Computing and Software)

McMaster University
Hamilton, Ontario

TITLE: Property Inference for Maple: An application of abstract interpretation

AUTHOR: Stephen A. Forrest, B.Math.(University of Waterloo)

SUPERVISOR: Dr. Jacques Carette

NUMBER OF PAGES: 1, 76

We present a system for the inference of various static properties from source code written in the Maple programming language. We make use of an abstract interpretation framework in the design of these properties and define languages of constraints specific to our abstract domains which capture the desired static properties of the code. Finally we discuss the automated generation and solution of these constraints, describe a tool for doing so, and present some results from applying this tool to several nontrivial test inputs.

In preparing this document I should like to acknowledge the patience, guidance and many helpful suggestions of my advisor, Jacques Carette. As well, I thank Kirsten Bos for continuing moral support throughout this long period of writing. Lastly, I should also thank Gordon Uszkay for his technical assistance in the preparation of this document.

Contents

Chapter 1

Introduction

Our goal is the static inference of properties from Maple code. The motivations we have for this effort are primarily the traditional arguments for static analysis: better program comprehension, automated error detection, and general desire for a saner world.

We are also curious to see what consequences the fact that Maple is a computer algebra system with some unusual and even unique features has on our analysis. Are there features shared by Maple and other symbolic computer algebra systems such as Mathematica and MuPAD, but not shared with any other programming languages? Might there be static properties entirely unique to Maple?

As well, as a dynamic language Maple is considerably more polymorphic than many authors of Maple programs realize. The “correct” semantics of a program may therefore differ considerably from the semantics the program author had in mind, and a tool like the one we propose might aid in detecting these cases of “inadvertent polymorphism”.

In a previous work ([2], with Jacques Carette), we attempted a very naïve type inference of Maple through a traversal of its library routines. It became clear through this analysis that more benefits could be obtained with the addition of some Maple-specific “opportunistic” knowledge.

Our framework for this analysis must be consistent, scaleable, and most importantly, correct. A major guiding factor in our approach is our desire to avoid the *toy problem syndrome*: a static analysis tool that as a consequence of design decisions is artificially limited in its scope or efficiency.

While in the interests of expediency we will obviously be bound by certain restrictions on the range of inputs that we can process, we wish to make our analysis as generic as possible. If no information about a particular component can be statically deduced, we would like to merely note this fact and move on, without this further limiting our analysis.

With these restrictions in mind, our ultimate choice is to employ various techniques from Data Flow Analysis to generate systems of constraints on values or state, whose correctness is assured using a framework of abstract interpretation. This gives us, for each property of interest, a convenient language of *constraints* in which to express the relationships between program points.

The work has considerable potential, not just for static analysis and overall code comprehension but also as as an enabler of automated program transformations. In particular, it would aid in the breadth and range possible for tools like partial evaluators [20, 4], type inferencers, and code generation.

For example, a significant problem faced in performing partial evaluation on a Maple procedure is resolving and residualizing a procedure call. We may be able to greatly simplify the specialized procedure if we know how many arguments were supplied in calling it; however, in Maple this knowledge generally requires static analysis to compute.

With a longer-term view, one could combine several of these static analyses in an effort to do static type inference on the code. While the constraint languages here are simple by design, the language of inferred types would be complex enough to capture all or most of the information present in each quantity.

In our presentation here, we are guided by the "separation of concerns" described by Cousot [5], who argued for clear distinctions between identification of the properties one wishes to estimate, the lattices employed in such estimations, the specification for a solution strategy, and the details of its actual implementation.

Chapter 2

Overview of Maple

We have opted to direct our efforts here specifically upon the Maple programming language. Many of the specific details that will interest and challenge us are not especially well-known outside Maple circles, so it is necessary to provide a brief survey of the language itself.

Our purpose here is threefold: to introduce and give a feeling for the language we shall be tackling to readers unfamiliar with the Maple programming language, to provide some motivation for our choice of properties to analyze in Chapter 4, and to chronicle the more peculiar aspects of the language that will present unique impediments to our analyses. For a more detailed discussion of the semantics of Maple, the interested reader should refer to [18].

In the examples below, we shall employ the convention that input to Maple is prefaced with a > prompt, and the corresponding output is shown on the following line.

In the following, "Maple" will usually refer to the Maple programming language, and less often to the commercial software package implementing said language. In general, the details discussed are independent of software version numbers, but whenever this is not the case, the version referenced is Maple 10.

Maple is an interpreted, untyped, procedural language with lexical scoping and first-class procedures.

Many of the analyses we wish to attempt are complicated by the particular semantics of Maple. Some of these, such as basic control structures, are shared with many other programming languages. Other features, such as support for arbitrary-precision integers, are shared with a much smaller set of other programming languages. Still other features, such as a symbolic default value for variables, are specific to a CAS or to Maple alone. Following is a description of some key features.

2.1 Data types

2.1.1 Base types

As one would expect from a mathematical programming language, Maple has a rich variety of base types. These include arbitrary-precision integers, rationals, hardware and software floating-point numbers, and complex numbers over all the previously-mentioned domains. Arbitrary-precision floating-point computations are possible through software.

Here we see an exact computation with rational numbers:

```
> 2^65 * (1/3^42) + 7;
80282641206800561769 5/109418989131512359209
```

Arithmetic with integers and rationals is exact by default; floating-point computations are performed only if specially requested or if a floating-point number is introduced into the computation. The same example with `7.0` illustrates the significance a single floating-point number has on the outcome of the computation:

```
> 2^65 * (1/3^42) + 7.0;
7.337176284
```

In addition to numeric data, strings are a base type with and usual string operations (concatenation, subselection, pattern-matching, etc.)

2.1.2 Variables

Variables in Maple have a dual role as containers and data. If a variable name is not initialized, but used, its evaluated value is merely *itself*: or, more correctly, a symbolic reference to its own name. One might view such symbols as another basic data type.

This example shows that `a`, a variable with no past history, may be freely used as a symbol as part of an expression assigned to another variable.

```
> b := 2*a;
b := 2*a

> 3 * b^2
12*a^2
```

Since any fragment of code may alter the value of a global variable at any time, we cannot assume that state remains constant when evaluating an expression. Should a in the above example later be assigned a value, say 7, the variable b will afterwards evaluate to 14, not 2*a, even though no further writes to b have been made..

2.1.3 Container types

Maple has a wide variety of container types for data. These include *lists, sets, expression sequences*, *function applications*, as well as *hash tables*, arrays, vectors, and matrices.

It will be helpful to begin with the more familiar datatypes. A list is simply an arbitrary-sized ordered sequence of Maple objects. Syntactically, a list is a comma-delimited sequence of objects enclosed by opening and closing brackets. Lists are ordered and duplicate elements are allowed. A list may contain any Maple value, including other lists:

```
> [1, [4, 3, [7, 7]], 6];
[1, [4, 3, [7, 7]], 6]
```

Sets are similar to lists, but differ in that they are unordered and duplicate elements are removed. The ordering imposed by Maple is arbitrary, based on an element's memory address, and is also session-specific. Syntatically, a set is a comma-delimited sequence of objects enclosed by opening and closing curly braces:

```
> {1, 2, {1,2,3}, {2,3,1}, 2};
{1, 2, {1, 2, 3}}
```

We proceed now to a less-familiar datatype. An expression sequence (which we shall sometimes abbreviate as *expseq*) is an arbitrarily-sized list of values. Its values are syntactically delimited with commas; unlike lists and sets, no left and right braces are necessary.

The sequence of comma-delimited values from which an expseq is built may be any values, including other expseqs. However, expseqs are self-flattening: that is, the result of concatenating two expseqs is a single non-nested expseq:

```
> 1, (4, 3, (7, 7)), 6;
1, 4, 3, 7, 7, 6
```

There is a unique expression sequence of length 0; it is accessible by the special name `NULL`.

```
> NULL, (1, 2), NULL, 3;
1, 2, 3
```

In addition to being self-flattening, another notable feature of expression sequences is *automatic cast to value*: an expression sequence of length 1 automatically evaluates to its only element. Put another way, an object whose only potential length as an expression sequence is 1 is not an expression sequence.

(Observe that because of self-flattening and automatic cast to value, `NULL` is a left and right identity for the `,` operator.)

We can therefore partition the set of Maple values such that every value v is either an *expression* or an *expseq*, and in the latter case either $v = \texttt{NULL}$ or the length of expseq v is ≥ 2.

At this point, it is helpful to return to lists and sets. We spoke of lists and sets as containing values: to be precise, a list or set holds only a single value, which is an expseq. Lists and sets are merely wrappers; however, because they are expressions, they may be nested.

```
> [1, (4, 3, {7, 7}), 6;
[1, 4, 3, {7}, 6]
```

2.1.4 Function applications

Function applications deserve special attention as their behaviour is especially unique. A function application has the form `f(a)` where `f` is the function name and `a` is an expression sequence of arguments. Following Maple terminology we shall refer to `f` as the *zeroth operand* of `f(a)`.

The fact that `a` is an expression sequence provides another illustration of the ubiquity of expression sequences in Maple; one might say that all Maple functions take only a single argument (an expression sequence). Because the sequence of arguments is an expression sequence, it may be constructed dynamically, and so even the number of arguments passed in a function application may not be statically knowable. For example:

```
> S := 2,3,7,11;
2, 3, 7, 11
> igcd( S ); # same as igcd(2,3,7,11) because of previous assignment
1
```

We are not able to determine the number or type of arguments passed to `igcd` from the function application alone; to find this we must examine the preceding computation history. It is not only the input to function applications which is unusual. When the name `f` is assigned to a procedure (see 2.2), the procedure is applied the operands of the function application, and returns the result, as one would expect.

However, recall from 2.1.2 that an unassigned name has a symbolic default value. If `f` is an unassigned name, the application `f(a)` returns unevaluated, as a function data structure. If the symbol `f` later receives a value and this function data structure `f(a)` is re-evaluated, it will evaluate as a function application by applying the value of `f` to the argument `a`.

2.2 Procedures

As described in 2.1.4 when a function application is not a data structure as described above, it results in the execution of a *procedure*.

```
> p := proc(a,b,c,n)
      return(a^n + b^n = c^n);
  end proc:
> p(x,y,z,3)
x^3 + y^3 = z^3
```

A procedure body is a sequence of statements (see 2.4) from which a value is returned; if no value is provided explicitly, the system implicitly returns the last evaluated value.

Certain special variable names are usable in a procedure body and convey dynamic information about the function application that initiated the procedure. The name `args` provides the expression sequence of arguments passed into the procedure, while the name `nargs` gives the number of arguments provided (which, as mentioned in 2.1.4, may vary).

```
> p := proc()
      sprintf( "Number of arguments was: %d\n", nargs );
  end proc:
```

```
> p(1,2,4);
"Number of arguments was 3"
> p();
"Number of arguments was 0"
```

Procedures are lexically-scoped, with local variables in their own local scope. Local variables have a default symbolic value as do other Maple variables. These symbols, or structures containing or referencing them, may be freely passed out of the procedure: Maple thus supports *closures*.

The following example demonstrates Maple's support for closures, lexical scoping, and procedures as first-class values. The procedure in this example is a *counter generator* procedure: it initializes a local counter variable, then returns a *counter* procedure which contains a lexical reference to this counter, increments it upon each function call, and returns the result.

```
> CounterGenerator := proc(initialValue) local counter;
      counter := initialValue;
      proc()
          counter := counter + 1;
          counter;
      end proc:
  end proc:
> CounterProc := CounterGenerator(0):
proc() counter := counter + 1; counter end proc
> CounterProc();
1
> CounterProc();
2
```

In general procedures usable by Maple may be classified into three groups : *kernel*, *library*, and *user-defined*. They are distinguished thus:

- *Kernel procedures* or *built-in procedures* are those procedures that are not written in the Maple language but are built in to the system kernel itself. These typically involve core functionality.

- *Library procedures* are those written in the Maple language and included in the default Maple library.

- *User-defined procedures* are written in the Maple language but are not part of the default Maple library.

This distinction will become important later on, since we will be unable to perform analysis on built-in procedures as they are not written in the Maple language.

2.3 Typing

Though Maple is effectively an untyped language in its design, it does have some notion of type-checking. A particular subset of first-class values are *types*. (This idea of types as a subset of values is commonly called “Type:Type” in type theory literature [1] and has important consequences for type inference in the system, see [24, ch. 30]).

One may dynamically check whether a value conforms to a given type via a function call of the form `type(`*expression*`,` *type_expression* `)`, which returns a boolean result.

```
> type( 2/3, integer );
false
> type( ["pomme", "orange", "ananas"], list );
true
```

One can also put explicit type checks on the formal arguments to a user-defined procedure. These raise an exception when supplied with non-matching values.

```
> ThueMorse := proc(n::nonnegint)
        if n=0 then 0
        elif type(n, even) then procname( n/2 )
        else 1-procname( (n-1)/2 ) end if;
  end proc:
> ThueMorse(-1);
Error, (in ThueMorse) invalid input: ThueMorse expects its 1st
argument, n, to be of type nonnegint, but received -1
```

However, unlike in a statically-typed language, these checks are only performed at runtime at the moment of function application.

In this sense, Maple types can be regarded effectively as a class of dynamic predicates on values, and not types in the usual sense.

2.4 Statements and control structures

Many of the control structures of Maple are broadly similar to those encountered in other procedural languages and do not require special explanation. Maple has an assignment operator, if-then structures, loops, try/catch blocks, and error statements. We will discuss those aspects of these control structures which are Maple-specific or otherwise unusual.

2.4.1 Simultaneous assignment

In addition to assignment to a single variable, *simultaneous assignment* is supported; in the following example, the assignment to `a` and `b` is done *simultanously* in both steps:

```
> (a,b) := (1,2);
(a, b) := (1, 2)
> (a,b) := (b,a);
(a, b) := (2, 1)
```

Note that as a consequence, the value of `a` and `b` has been swapped without resorting to the use of temporary variables.

2.4.2 Loops

Maple supports two types of loops. It should be noted, however, that because of Maple's functional features (see 2.5) many computations that might otherwise employ loops instead utilize functional primitives for traversing expressions.
Two types of loop are supported:

- *For-from loop*: This loop takes a variable v and arguments (a, s, b, c) and steps from a to b by interval s, assigning each step value to v. This is essentially the standard procedural for-loop. (Note that `infinity` is an acceptable value for b, so the fact that b is given a value does not guarantee termination.)

- *For-in loop*: This loop takes a variable v and an value e and steps through the operands of e, assigning v to each in turn. This is the equivalent of the "foreach" loop in some programming languages.

Both loops have an optional dynamic condition which, if provided, is checked at the start of each loop iteration.

2.5 Functional features

While still fundamentally a procedural language, Maple has many functional features. These include first-class procedures, pattern matching, a `map` command for mapping over data structures, a functional `if` operator, functional operators for arithmetic operations, and library support for currying, composition, and λ-abstraction.

The following example computes the element maximum of two lists using `zip` and `max`:

```
> zip( max, [1, 7, 5, 9], [6, 6, 8, −1] );
[6, 7, 8, 9]
```

The following is a Maple implementation of the function `concatMap` from the Haskell Prelude (see [23]).

```
> concatMap := proc(f,a) local x; [seq(op(f(x)),x=a)] end proc:
```

The command `combinat:-partitions(n)` gives a list of all partitions of a positive integer n; in this sense a partition is a list $[a_1, \ldots, a_n]$ such that $a_i \in \mathbb{N}$ and $1 \leq a_i \leq n$ for $i = 1, \ldots, n$ and $a_1 + \cdots + a_n = n$. This example computes all partitions up to size 3:

```
> concatMap( combinat:-partitions , [1, 2, 3] );
[[1], [1, 1], [2], [1, 1, 1], [1, 2], [3]]
```

However, we should not overstate our case here: appearance notwithstanding, Maple is very far from the typical characterization of a functional language. Any procedure can alter global state at any time, should it opt to do so. An even stronger argument is that because variables can be passed around as symbols prior to receiving a value, and anything glimpsed *as a symbol* can be assigned, this means that an innocuous-looking function application with an unknown or dynamic function name has the capacity to alter the state of any of its arguments. Therefore we can not even always trust that any state changes will be confined to global (and not local) state.

Here is a simple example of a procedure that receives an argument and promptly attempts to write the value 2 to it.

```
> writeTwo := proc(s) if type(s, name) then assign(s,2) end if; end proc:
> writeTwo( freshSymbol );
> freshSymbol;
2
```

2.6 Evaluation levels

As the system permits the creation of deeply-nested expressions containing symbolic quantities, it is necessary to allow some flexibility on how expressions are evaluated. There are several tools for this purpose:

Maple has a *delay evaluation* operator: this is simply a *thunk*. If e would evaluate to an expression b, the result of evaluating a when surrounded by the delay operator will be simply a, rather than its evaluated result b. Additionally, there are ways of forcing custom evaluation levels. The command `eval` will evaluate the expression as far as it can. Though this is often necessary, it has the potential to become quite expensive.

```
> ToInert( x->3*x^2 );
  _Inert_PROC(
      _Inert_PARAMSEQ(
          _Inert_NAME("x")
      ),
      _Inert_LOCALSEQ(),
      _Inert_OPTIONSEQ(
          _Inert_NAME("operator"),
          _Inert_NAME("arrow")
      ),
      _Inert_EXPSEQ(),
      _Inert_STATSEQ(
          _Inert_PROD(
              _Inert_POWER(
                  _Inert_PARAM(1),
                  _Inert_INTPOS(3)
              ),
              _Inert_INTPOS(2)
          )
      ),
      _Inert_DESCRIPTIONSEQ(),
      _Inert_GLOBALSEQ(),
      _Inert_LEXICALSEQ()
  )
```

Figure 2.1: ToInert example, formatted for readability

2.7 Reflection tools

One of the features of Maple which is especially interesting and which will prove especially useful is its built-in support for *reflection*. This support is provided by a pair of functions which convert "live" code to a so-called *inert form*.

Concretely, the inert form is a data structure made of a series of nested function calls with symbolic function names. Because the names are symbols, the expression is *inert*, meaning it will not evaluate to anything other than its present value. This use of symbolic function applications as a customized data structure is a common pattern in Maple.

However, the key point is the data in the data structure. The symbols used in the inert form and the arrangement of function calls is a representation of the *abstract syntax tree* of the original expression from which the inert form was generated. In general, each function application in the inert form corresponds to a node in the abstract syntax tree.

The result is that we may freely transform a “live” expression into data which is organized in a consistent manner and which can be studied and manipulated without fear of accidentally triggering an evaluation. Once we are finished, the data may be freely transformed back into a “live” expression.

The symbols present as function names in inert form data structures all belong to a finite set of symbols: they all take the form `_Inert_foo` where `foo` is the name of data type or control structure. We may think of these as equivalent to labels on nodes in the AST which qualify which type of expression or statement we are examining.

We will refer to these `_Inert_foo` symbols as “inert tags.” The list of inert tags that may be produced by a call to `ToInert` is lengthy but finite. For a complete accounting, see Appendix A.

The built-in procedure `ToInert` transforms an input expression into an *inert form*. Following is a short example; for a lengthier one see figure 2.1.

```
> ToInert( [-1, 1] );
        _Inert_LIST(_Inert_EXPSEQ(_Inert_INTPOS(1), _Inert_INTNEG(-1)))
```

The inverse operation to `ToInert` is, naturally enough, `FromInert`, which transforms an inert form into a “live” Maple object.

```
> FromInert(_Inert_SUM(_Inert_NAME("a"), _Inert_INTNEG(5)));
a - 5
```

The inert form shall be an essential low-level tool in our analysis. The abstract syntax tree as provided by `ToInert` will be the basic unit upon which all our analyses will be performed. For this reason, it is `ToInert` which we shall be using most frequently: there are however some occasions on which we will be grateful for the capability of transforming code in either direction.

Chapter 3

Constraint-based Data Flow Analysis and Abstract Interpretation

In this chapter we present the theoretical underpinnings of our analysis. Here our discussion is generic and makes no reference to Maple; we intend to motivate our later exposition and introduce important concepts and terminology which will be used later.

3.1 Constraint-based Data Flow Analysis

Data Flow Analysis is a widely-studied and well-established branch of static analysis, in heavy use in both the academic and commercial field. Here we briefly summarize a particular technique, data flow analysis utilizing a constraint-based approach, which we shall make use of later.

Our somewhat informal presentation is loosely based on [22, pp. 8-10, 41-43]. See also [28] for a good presentation of Reaching Definitions and other related Data Flow analyses.

3.1.1 System of constraints

Given an input program p, we first obtain its abstract syntax tree $\mathbf{AST}(p)$. We are typically interested in a specific subset of the nodes of $\mathbf{AST}(p)$, specific to our particular analysis. To each of these "nodes of interest" we affix a unique label drawn from some fresh set of symbols. Let $\mathbf{Lab}$ denote the set of labels; clearly $\mathbf{Lab}$ will be finite because $\mathbf{AST}(p)$ is finite.

At this point we have in mind some lattice $(L, \sqsubseteq)$ of values. It is our goal to construct a map $\varphi : \mathbf{Lab} \to L$; that is, to associate with every program point $\ell \in \mathbf{Lab}$ of interest a value in L.

Let us define the set M as follows:

- If $x \in \mathbf{Lab}$ or $x \in L$, then $x \in M$.
- If $f : L^n \to L$ is a monotone function and $x_1, \ldots, x_n \in M$, then $f(x_1, \ldots, x_n) \in M$.

The central idea is to generate, in a way specific to the input language and lattice L, a system of *constraints*. These are predicates of the form $\ell \sqsupseteq x$ where $\ell \in \mathbf{Lab}$ and $x \in M$. (Without loss of generality we take $\ell \sqsupseteq x$; we could have also chosen $\ell \sqsubseteq x$, as long as the direction is consistent across all constraints.)

Given any two constraints $\ell \sqsupseteq x$ and $\ell \sqsupseteq y$ for some $\ell \in \mathbf{Lab}$ and $x, y \in M$, we can unify these into a single constraint with the lattice join operator; that is:

$$(\ell \sqsupseteq x) \wedge (\ell \sqsupseteq y) \implies \ell \sqsupseteq (x \sqcup y)$$

After all these unifications have been performed, we end with a system of constraints of the form

$$\begin{array}{ccc} \ell_1 & \sqsupseteq & F_1(\ell_1, \ldots, \ell_N) \\ \vdots & \vdots & \vdots \\ \ell_N & \sqsupseteq & F_N(\ell_1, \ldots, \ell_N) \end{array}$$

where N is the (finite) size of $\mathbf{Lab}$, and the ℓ_i are all distinct.
We may write this as $\vec{\ell} \sqsupseteq F(\vec{\ell})$ for $F \in Ł^N \to L^N$.

3.1.2 Fixed points

The fact that F is monotone and $F(\vec{\bot}) \sqsupseteq \vec{\bot}$ tells us that $F^{n+1}(\vec{\bot}) \sqsupseteq F^n(\vec{\bot})$ for any $n \in \mathbb{N}$.

Definition 1 *Let $(L, \sqsubseteq, \sqsupseteq)$ be a lattice. A sequence $\{a_i\}_{i=1}^{\infty}$ with $a_i \in L$ is said to be an* ascending chain *if $a_n \sqsubseteq a_{n+1}$ for all $n \in \mathbb{N}$.*
The L satisfies the Ascending Chain Condition *if for every ascending chain $\{a_i\}_{i=1}^{\infty}$ there is some $N \in \mathbb{N}$ such that $a_i = a_j$ for $i, j > N$.*

If L is finite or satifies the Ascending Chain Condition, there exists some $n \in \mathbb{N}$ such that $F^{n+1}(\vec{\bot}) = F^n(\vec{\bot})$.

This is in fact also a least solution to our constraint problem $\vec{\ell} \sqsupseteq F(\vec{\ell})$. Therefore, starting from $\vec{\ell} = \vec{\bot}$ we may generate a solution in n steps by successive application of F to the incremental result, provided of course that the convergence criteria hold. We will later discuss what to do if they do not hold.

3.2 Abstract interpretation

Abstract Interpretation [9, 11] is a general theoretical framework for the sound approximation of program semantics. Its generality and applicability to many different domains makes it particularly well-suited for use as a program analysis methodology.

Here, we provide a brief introduction to the field. The interested reader may learn more from the many papers of P. Cousot ([14, 6] being particularly relevant for our purposes). This overview has been adapted from a shorter one by Jacques Carette and this author (see [3]); this in turn was greatly influenced by the pleasant introduction [26] by Mads Rosendahl, and by David Schmidt's lecture notes [27].

While the many approaches that fall under the rubric of "abstract interpretation" differ in both their theoretical underpinnings and the subjects of their analyses, there is a single unifying idea.

Given a program p, we assign two distinct semantics to p. The first is the "usual" or *concrete* semantics, which models the runtime behaviour of p. The other semantics, the *abstract* semantics, is typically chosen because it is easier to compute or reason with. We require that our two interpretations of p "agree" in a sense, so that we may answer certain questions about the runtime behaviour of p (i.e. its concrete semantics) by examining its abstract semantics and translating this back to the concrete world.

More formally, our assignment of semantics to p is an *interpretation*. We therefore have, two interpretations $I_1[\![p]\!]$ and $I_2[\![p]\!]$, where $I_1[\![p]\!]$ is a *concrete interpretation* and $I_2[\![p]\!]$ an *abstract interpretation*.

The entire aim of the technique of abstract interpretation is the judicious selection of $I_1[\![p]\!]$ and $I_2[\![p]\!]$ in such a way that $I_1[\![p]\!]$ models the "real world", $I_2[\![p]\!]$ is easier to handle, and the two are related in such a way that we may infer facts about $I_1[\![p]\!]$ from $I_2[\![p]\!]$.

3.3 Example: Rule of sign

To make this discussion less "abstract," let us begin with a standard example, the *Rule of sign* (for an early description, see [29]). Consider a simple expression language given by the grammar

$$e ::= n \mid e + e \mid e * e$$

(In the above, n is a placeholder for all integers $n \in \mathbb{Z}$.)

The standard interpretation is usually given as

$$\begin{array}{cc} E[\![e]\!] \ : \ \mathbb{Z} & E[\![e_1 + e_2]\!] = E[\![e_1]\!] + E[\![e_2]\!] \\ E[\![n]\!] = n & E[\![e_1 * e_2]\!] = E[\![e_1]\!] * E[\![e_2]\!] \end{array}$$

We wish to be able to predict the sign of an expression whenever possible, by using only the signs of the constants in the expression.

Our chosen abstract domain will allow us to distinguish between expressions that are constantly zero, positive or negative. In fact, however, this is not enough: if we add a positive integer to a negative integer, we cannot know the sign of the result (without actually performing the addition). So we also give ourselves a value to express this uncertainty, and denote that all we know is the result is a 'number'.

Taking **Sign** $= \{\mathsf{zero}, \mathsf{pos}, \mathsf{neg}, \mathsf{num}\}$, we can define an "abstract" version of addition and multiplication on **Sign**:

$\oplus$: **Sign** $\times$ **Sign** $\rightarrow$ **Sign**

$\oplus$	neg	zero	pos	num
neg	neg	neg	num	num
zero	neg	zero	pos	num
pos	num	pos	pos	num
num	num	num	num	num

$\otimes$: **Sign** $\times$ **Sign** $\rightarrow$ **Sign**

$\otimes$	neg	zero	pos	num
neg	pos	zero	neg	num
zero	zero	zero	zero	zero
pos	neg	zero	pos	num
num	num	zero	num	num

Using these operators, we can define an *abstract interpretation* for expressions as:

$$\begin{array}{cc} A[\![e]\!] \ : \ \textbf{Sign} & A[\![e_1 + e_2]\!] = A[\![e_1]\!] \oplus A[\![e_2]\!] \\ A[\![n]\!] = \mathsf{sign}(n) & A[\![e_1 * e_2]\!] = A[\![e_1]\!] \otimes A[\![e_2]\!] \end{array}$$

where sign(x) = **if** $x > 0$ **then** pos **else if** $x < 0$ **then** neg **else** zero.

How are the interpretations $E[\![x]\!]$ and $A[\![x]\!]$ related? Formally, we can describe the relation between them as follows (and this is typical):

$$\gamma : \textbf{Sign} \to \mathcal{P}(\mathbb{Z}) \setminus \emptyset \qquad\qquad \alpha : \mathcal{P}(\mathbb{Z}) \setminus \emptyset \to \textbf{Sign}$$

$$\begin{aligned}\gamma(\mathsf{neg}) &= \{x \mid x < 0\}\\ \gamma(\mathsf{zero}) &= \{0\}\\ \gamma(\mathsf{pos}) &= \{x \mid x > 0\}\\ \gamma(\mathsf{num}) &= \mathbb{Z}\end{aligned} \qquad \alpha(X) = \begin{cases}\mathsf{neg} & X \subseteq \{x \mid x < 0\}\\ \mathsf{zero} & X = \{0\}\\ \mathsf{pos} & X \subseteq \{x \mid x > 0\}\\ \mathsf{num} & \text{otherwise}\end{cases}$$

The (obvious) relation between γ and α is:

- For all $s \in$ **Sign**, we have $\alpha(\gamma(s)) = s$.
- For all $X \in \mathcal{P}(\mathbb{Z}) \setminus \emptyset$, we have $X \subseteq \gamma(\alpha(X))$.

γ is called a concretization function, while α is called an abstraction function. Note these functions allow a much simpler definition of the operations on signs:

$$\begin{aligned}s_1 \oplus s_2 &= \alpha(\{x_1 + x_2 \mid x_1 \in \gamma(s_1) \sqcap x_2 \in \gamma(s_2)\})\\ s_1 \otimes s_2 &= \alpha(\{x_1 * x_2 \mid x_1 \in \gamma(s_1) \sqcap x_2 \in \gamma(s_2)\})\end{aligned}$$

3.4 Sound approximations

From this we get the very important relationship between the two interpretations:

$$\forall e.\{E[\![e]\!]\} \subseteq \gamma(A[\![e]\!])$$

In other words, we can safely say that the abstract domain provides us with a *correct* approximation to the behaviour in the concrete domain. This relationship is often called a *safety* or *soundness* condition. So while a computation over an abstract domain may not give us very useful information (think of the case where the answer is num), it will never be incorrect, in the sense that the true answer will always be contained in what is returned. More generally we have the following situation:

3.4.1 Galois connections

Definition 2 *Let $\langle C, \sqsubseteq \rangle$ and $\langle A, \sqsubseteq \rangle$ be complete lattices, and let $\alpha : C \to A$, $\gamma : A \to C$ be monotonic and ω-continuous functions. If for all $a \in A, c \in C$ we have the condition that*

$$c \sqsubseteq_C \gamma(\alpha(c)) \Longleftrightarrow \alpha(\gamma(a)) \sqsubseteq_A a$$

then we say we have a Galois connection*. If in fact for all $a \in A, c \in C$ we have the stronger condition:*

$$c \sqsubseteq \gamma(\alpha(c), \quad \alpha(\gamma(a)) = a$$

then we say we have a Galois insertion.

The reader is urged to read [17] for a complete mathematical treatment of lattices and Galois connections. The main property of interest is that α and γ fully determine each other. Thus it suffices to give a definition of $\gamma : A \to C$; in other words, we want to name particular subsets of C which reflect a property of interest. More precisely, given γ, we can compute α via $\alpha(c) = \sqcap\{a \mid c \sqsubseteq_C \gamma(a)\}$, where $\sqcap$ is the meet of A.

Given this, we will want to synthesize abstract operations in A to reflect those of C; in other words for a continuous lattice function $f : C \to C$ we are interested in $\tilde{f} : A \to A$ via $\tilde{f} = \alpha \circ f \circ \gamma$. Unfortunately, this is frequently too much to hope for, as this can easily be uncomputable. However, this is still the correct goal:

Definition 3 *For a Galois connection (as above), and functions $f : C \to C$ and $g : A \to A$, g is a* sound approximation *of f if and only if*

$$\forall c. \alpha(f(c)) \sqsubseteq_A g(\alpha(c))$$

or equivalently

$$\forall a. f(\gamma(a)) \sqsubseteq_C \gamma(g(a)).$$

Then we have that (using the same language as above)

Proposition 1 *g is a sound approximation of f if and only if $g \sqsubseteq_{A \to A} \alpha \circ f \circ \gamma$.*

How do we relate this to properties of programs? To each program transition from point p_i to p_j, we can associate a *transfer function* $f_{ij} : C \to C$, and also an abstract version

$\tilde{f}_{ij} : A \to A$. This defines a computation step as a transition from a pair (p_i, s) of a program point and a state, to $(p_j, f_{ij}(s))$ a new program point and a new (computed) state. In this we are performing a kind of *pseudo-evaluation*,

We always restrict ourselves to *monotone* transfer functions, i.e. such that

$$l_1 \sqsubseteq l_2 \implies f(l_1) \sqsubseteq f(l_2)$$

which essentially means that we never lose any information by approximating. This is not as simple as it sounds: features like *uneval quotes*, if treated naïvely, could introduce non-monotonic functions.

3.4.2 Return to data flow analysis

We return momentarily to the discussion in Section 3.1 to comment on the implications of Galois connections for this approach. In Section 3.1, we saw that for L a finite lattice one could compute the fixed point of $\vec{\ell} \sqsupseteq F(\vec{\ell})$ simply by iterating F, starting from $\vec{\ell} = \vec{\bot}$ until a fixedpoint was reached.

It is worth noting now that same is possible with a Galois connection. Briefly, if we take G to be an operator in the constraint language which is the *concrete* analogue of the language described in Section 3.1, our concretization and abstraction functions will induce a relation

$$\vec{\ell} \sqsupseteq (\vec{\alpha} \circ G \circ \vec{\gamma})\vec{\ell}$$

As $\vec{\alpha} \circ G \circ \vec{\gamma}$ is also monotonic, it too has a least fixed point that can be obtained by simple iteration when the abstract lattice is finite.

How does $\vec{\alpha} \circ G \circ \vec{\gamma}$ compare to F? We have placed no conditions on F other than monotonicity; since it simply represents some attempt to approximate the operational semantics of the program, there is no guarantee that the designer "thought of everything."

On the other hand, $\vec{\alpha} \circ G \circ \vec{\gamma}$ is *defined* entirely by the operational semantics of the program: it cannot therefore fail to be ideal. In fact we have $\vec{\alpha} \circ G \circ \vec{\gamma} \sqsubseteq F$ in general. The condition $\vec{\alpha} \circ G \circ \vec{\gamma} = F$, sometimes provable, indicates that the analysis F is *optimal* for its abstract domain.

3.5 Collecting semantics

In general, we are interested in *execution traces* or *collecting semantics*, (see [15, 22]) which are (possibly infinite) sequences of the transitions discussed in Section 3.4.1.

A trace corresponds to one particular execution path through the procedure being analyzed; in some sense, it is a projection of the property of interest into some idealized world in which we may magically obtain information which in reality is dynamic and inaccessible. But because *every possible execution* of the program is captured by some trace, we are assured of correctness.

Thus, a set of traces is frequently chosen as the *concrete semantics* when establishing the correctness of an analysis through a Galois connection.

A typical example is the classical analysis Reaching Definitions (see [22, p. 15]). Here, the goal is to determine, for a program point ℓ, and for each variable x, the label of the most recent assignment to x.

Of course in general we will not find a single answer, as there may have been previous conditional branches prior to ℓ in which x was written. Our abstract semantics is then $\mathcal{P}(\mathbf{Var} \times \mathbf{Lab})$. But because a trace corresponds to one particular execution path, there is always a unique past computation history: that is, $tr \in (\mathbf{Var}, \mathbf{Lab})^*$ (the $*$ is a *Kleene star*). Our concrete semantics domain is then simply $\mathcal{P}(\mathbf{Var} \times \mathbf{Lab})^*$.

3.6 Widening and narrowing operators

We have mentioned several times (such as in Section 3.1) the condition that our abstract domain either be finite or satisfy the Ascending Chain Condition to guarantee convergence upon a fixed point. We now examine what recourse is left to us when that it not the case. Here we summarize the presentation in [22, pp. 222-230]. See also [15] for a concise summary.

3.6.1 Widening operators

Let L be a complete lattice and $f : L \to L$ be a monotone function. We define the following sets:

$$\begin{array}{rcll} \mathrm{Fix}(f) & = & \{x : f(x) = x\} & (x \text{ is a fixedpoint of } f) \\ \mathrm{Red}(f) & = & \{x : f(x) \sqsubseteq x\} & (f \text{ is reductive at } x) \\ \mathrm{Ext}(f) & = & \{x : f(x) \sqsupseteq x\} & (f \text{ is extensive at } x) \end{array}$$

We define $\mathrm{lfp}(f)$, the *least fixed point of* f to be the greatest lower bound of $\mathrm{Fix}(f)$ in L, and $\mathrm{gfp}(f)$, the *greatest fixed point of* f to be the least upper bound of f in L.

Tarski's Fixed Point Theorem (see [10] for a constructive proof due to Cousot) imposes the condition that

$$\begin{array}{lll} \mathrm{lfp}(f) & = \bigsqcap \mathrm{Fix}(f) & = \bigsqcap \mathrm{Red}(f) \\ \mathrm{gfp}(f) & = \bigsqcup \mathrm{Fix}(f) & = \bigsqcup \mathrm{Ext}(f) \end{array}$$

and it may be shown that

$$f^n(\bot) \sqsubseteq \mathrm{lfp}(f) \sqsubseteq \mathrm{gfp}(f) \sqsubseteq f^n(\top)$$

However, all of the above inclusions may be strict for all $n \in \mathbb{N}$.

Such problems may be dealt with by the judicious use of widening operators [8, 9] which ensure termination and convergence upon a safe upper approximation of $\mathrm{lfp}(f)$.

Definition 4 *Let L be a complete lattice. An operator $\check{\sqcup} : L \times L \to L$ is an upper bound operator if $x_1 \sqsubseteq (x_1 \check{\sqcup} x_2)$ and $x_2 \sqsubseteq (x_1 \check{\sqcup} x_2)$. That is, $\check{\sqcup}$ is guaranteed to be as large or larger than both its arguments.*

Now, let $\{x_n\}_{n=1}^{\infty}$ be a sequence in L and take any $\phi : L \times L \to L$. We can define a new sequence $\{x_n^{\phi}\}_{n=1}^{\infty}$ as follows:

$$x_n^{\phi} = \begin{cases} x_0 & \text{if } n = 0 \\ \phi(x_{n-1}^{\phi}, x_n) & \text{if } n > 0 \end{cases}$$

It can be shown that whenever $\check{\sqcup}$ is an upper bound operator that $\{x_n^{\check{\sqcup}}\}_{n=1}^{\infty}$ is an ascending chain, and $x_n^{\check{\sqcup}} = \bigsqcup \{x_1, \ldots, x_n\}$ for all $n \in \mathbb{N}$.

(Observe also, incidentally, that the join operator $\sqcup$ acts as an identity on any sequence that was already an ascending chain.)

Definition 5 *An upper bound operator $\nabla : L \times L \to L$ is an widening operator if for every sequence $\{x_n\}_{n=1}^{\infty}$ the ascending chain $\{x_n^{\nabla}\}_{n=1}^{\infty}$ eventually stabilizes.*

An example of a nontrivial widening operator is taken from [22, p. 223] on the lattice $\mathbb{I}(\mathbb{Z})$. We pick some $s \in \mathbb{I}(\mathbb{Z})$ and define $\check{\sqcup}_s : \mathbb{I}(\mathbb{Z}) \times \mathbb{I}(\mathbb{Z}) \to \mathbb{I}(\mathbb{Z})$ as follows:

$$x \check{\sqcup}_s y = \begin{cases} x \sqcup y & \text{if } x \sqsubseteq s \text{ or } y \sqsubseteq x \\ [-\infty..\infty] & \text{otherwise} \end{cases}$$

If we have an infinite ascending chain like

$$[0..0], [0..1], [0..2], [0..3], [0..4], \ldots$$

and we ensure the endpoints of s are finite, say $s = [0..2]$, then the widening operator $\check{\sqcup}_{[0..2]}$ transforms this into the ascending chain

$$[0..0], [0..1], [0..2], [-\infty..\infty], [-\infty..\infty], \ldots$$

which of course has stabilized.

Given a widening operator $\nabla : L \times L \to L$ and a monotone function $f : L \to L$, we can then define a sequence f_{∇}^{n} defined by

$$f_{\nabla}^{n} = \begin{cases} \bot & \text{if } n = 0 \\ f_{\nabla}^{n-1} & \text{if } n > 0 \text{ and } f(f_{\nabla}^{n-1}) \subseteq f_{\nabla}^{n-1} \\ f_{\nabla}^{n-1} \nabla f(f_{\nabla}^{n-1}) & \text{otherwise} \end{cases}$$

Speaking roughly, the second case $f(f_{\nabla}^{n-1}) \subseteq f_{\nabla}^{n-1}$ tells us that we have overshot $\mathrm{lfp}(f)$, and that we should stop here. In fact we define $\mathrm{lfp}_{\nabla}(f)$ to be the f_{∇}^{m} for the first m that stabilized the chain, and we have

$$\mathrm{lfp}(f) \sqsubseteq \mathrm{lfp}_{\nabla}(f)$$

giving us our desired safe approximation.

3.6.2 Narrowing operators

The use of widening operators certainly moves us from the realm of danger into safe territory, but it is perhaps too safe. Perhaps, having leaped ahead past the least fixedpoint, we can take a few steps back to refine our approximation.

Recall that the triggering condition for stabilization was the fact that $f(f_\nabla^{n-1}) \subseteq f_\nabla^{n-1}$, which is stating that the function f has become *reductive*. To employ this idea naïvely and apply f repeatedly to $\mathrm{lfp}_\nabla(f)$ would be unwise, as we have no idea whether that will stabilize. The answer is a *narrowing operator*, defined as follows:

Definition 6 *An operator $\Delta : L \times L \to L$ is a narrowing operator if*

- *whenever $x \sqsubseteq y$, then $x \sqsubseteq (x \Delta y) \sqsubseteq y$ for all $x, y \in L$, and*
- *for every descending chain $\{x_n\}_{n=1}^\infty$, the descending chain $\{x_n^\Delta\}_{n=1}^\infty$ stabilizes.*

With this narrowing operator we can then define a further approximation of f which will stabilize downwards and provide a worthy approximation of $\mathrm{lfp}(f)$.

$$[f]_\Delta^n = \begin{cases} \mathrm{lfp}_\nabla & \text{if } n = 0 \\ [f]_\Delta^{n-1} \Delta f([f]_\Delta^{n-1}) & \text{if } n > 0 \end{cases}$$

The sequence $[f]^n$ will stabilize for some value $m' \in \mathbb{N}$. We define

$$\mathrm{lfp}_\nabla^\Delta(f) = [f]^{m'}$$

This is our final approximation for f.

3.6.3 Comparison of approaches

The question of the overall utility of widening and narrowing operators is an interesting one. Some have argued [12, 13] that the results of using such operators can be replicated or approximated merely by avoiding infinite lattices or lattices without the ascending chain condition in the first place, and that use of narrowing and widening operators can sometimes lead to arbitrary loss of information.

In [13], Cousot contrasts the use of widening and narrowing operators with the "Galois connection approach" of choosing an abstract domain of sufficient simplicity to avoid infinite chains. He argues that the widening/narrowing approach may give better results, though he suggests that in many cases widening and narrowing operators are best used in connection with a Galois connection approach which performs some initial simplification of the problem domain.

Chapter 4

Properties and their domains

4.1 Introduction

Our goal is the static inference of various properties from Maple source code based on our knowledge of Maple's operational semantics. Given our previous discussion about the unique aspects of Maple, one might expect this distinctness to guide us in our choice of properties of interest, and indeed that is the case.

Since we have opted to place ourselves in an abstract interpretation framework, we wish to investigate those properties which can be be *approximated* using *complete lattices*. As it turns out, this requirements is easy to satisfy in various ways.

Many of the classical intraprocedural data-flow analyses described in ([22, pp. 37-52]), such as Available Expressions, Live Variables, and Very Busy Expressions do not fit naturally with Maple because of its special semantics, particularly its use of symbolic variables. We will, however, make extensive use of a modified form of the Reaching Definitions Analysis.

4.2 Properties of interest

We can divide the properties of interest into those that are *state-based*, those that are *program-based*, and those that are *value-based.*

A state-based property associates a *program point* with an approximation of program state; computing such a property amounts to a *pseudo-evaluation* of the program statements projected into the space of the property of interest. Statements or other operations affecting state, such as assignments, induce *transfer functions* between properties.

A program-based property is an property of the program or procedure which depends on the source code (specifically the abstract syntax tree) in some direct manner. Some examples of this category might be various properties of program blocks, such as a property measuring the number of iterations of a for loop.

A value-based property is one which associates a program point with an approximation of an expression *value*. An earlier example illustrating this in the field of abstract interpretation is the classical sign approximation, where we seek to model the sign of an arithmetic quantity. Here, functions and operations on values induce transfer functions between properties.

4.2.1 State-based properties

Reaching Definitions

*Reaching Definition*s, occasionally called *Reaching Assignments*, is a classical analysis (see [22, pp. 4-10, 41-44] or [28, p. 26]), the goal of which is to determine, at every program point, the last assignment made to each "currently active" variable.

Such an analysis conveys a wealth of information about the state and data flow of the program in question, but we shall need still more (as per the next item).

The Reaching Definitions analysis is closely tied to the *Use-Definition* and *Definition-Use* chains, which link variable instances to their prospective definitions and vice-versa. In fact, these chains may be easily contructed from a Reaching Definitions analysis; see [22, pp. 50-52].

Reaching Contexts

Reaching Contexts is a generalization of Reaching Definitions. At some program point, we seek to determine, for each "currently active" variable, the last *statement context* which affected our knowledge about that variable.

This is a somewhat subtler notion than state alone, and will help to discuss what this means for specific types of statements. Consider the following program:

```
p := proc(N::integer) local n, i, s;
     if [N > 0]^1 then n := [N]^2; else n := −[N]^3; end if;
     [s := 0;]^4
     for i from 1 to [n]^5 do
          [s := [s]^7 + i^2;]^6
     end do;
end proc:
```

Observe that label 2 corresponds to the first branch of the if-block. The evaluation of the condition N>0 did not change the state of the variable N, but it does change our knowledge: within that branch N>0 holds.

Similarly, consider the variable s at position 7 within the for-loop. A Reachings Definitions analysis would determine that s last received its value at either label 4 or label 6 (in a previous iteration of the loop). It is perhaps more helpful merely to recognize, at the beginning of a loop iteration, that s is a variable which may be transformed by the loop.

Informally, the goal of the Reaching Contexts is to determine the last assignment, loop, or if structure which may have affected our knowledge about a variable. We are interested only in assignments which write to v, if conditions in which v appears, and loops in which v may be transformed. We shall make this definition precise in Section 4.3.1.

4.2.2 Program-based properties

Number of variable reads

We wish to estimate how many times each of the ‘active variables’ have been read. For each label ℓ corresponding to a program block and each local variable v, can we tell the number of times v has been read in the scope of ℓ?

Number of variable writes

This is a natural (semantic) dual to the Number of variable reads analysis; it is, however, operationally independent so we have opted not to try to capture both within a single property.

Number of loop steps

Given a program point ℓ corresponding to a loop, how many times in an execution of the program will the loop ℓ iterate before terminating?

One thing to clarify is that we wish to know how many times the loop ℓ will iterate, not how many times its body will be executed. This becomes important in the case of nested loops, where the inner loop is repeatedly re-executed. For example:

```
s := 0;
[for i from 1 to 3 do [for j from 1 to 5 do s := s+i+j; od;]^2 od;]^1
```

Here, our solution for label 2 is 5, not 15, even though the statement body executes 15 times.

The loop-step analysis provide a good example of the interdependence of many of these properties: clearly, the number of steps of a loop is dependent on the expressions or numerical ranges being traversed, which are the subject of other properties.

4.2.3 Value-based properties

Surface type

An obvious candidate for a property of interest is the *type* of an value; indeed, type inference is a frequent goal of static analysis. Rather than attempting to use Maple's own rather idiosyncratic system for typing, or inventing our own system, we will allow ourselves to be guided by Maple's own *inert form*. We will construct a quantity we call a *surface type* based on the inert form.

As described in section 2.7, any Maple value `v` may be converted to an inert form with the command `ToInert(v)`. This inert form is a function data-structure: the root of this structure (in Maple parlance, the zeroth operand) is always one of the inert tags (see 2.7). This provides us with a clear and consistent concrete semantics for classifying values into something resembling types. We say that value `v` has *surface type* `t` if the result of executing `op(0, ToInert(v))` is `t`.

```
> op(0, ToInert( x = 2*y ) );
_Inert_EQUATION
```

Our aim is therefore to take a program point ℓ corresponding to a value and determine the surface type of the dynamic value corresponding to ℓ.

The motivation for the name “surface type” is that the type captures only the root of the abstract syntax tree: the type of an expression does not capture any information about leaves or internal nodes.

Expression sequence length

Here we are really doing two inferences at once: given a label ℓ corresponding to a value, we seek to determine whether the value is an expression or expression sequence (refer to 2.1.3), and if the latter, what its length is.

From Maple’s semantics, we know that expseqs behave quite differently in many contexts than other value, so it is important to know whether a given value is an expression sequence. This becomes particularly important in the analysis of function applications, for reasons described in 2.1.3.

Number of operands

In the event that program point ℓ is an expression, how many operands does this expression have? That is, if it is a list or set, what is its size? If it is a function application, how many operands are present? In most cases this is equivalent to asking about the number of immediate children of the node corresponding to ℓ in the abstract syntax tree.

This analysis is inspired by the Maple command `nops` which performs an identical function on concrete inputs.

Sometimes this information comes “for free” from the expression sequence length analysis: for example, in the input [1,2,3], the expression sequence 1,2,3 is part of the input, and will therefore be “seen” by the Expression sequence length analysis. However, this is not always the case.

Because we must first know that the value ℓ *is* an expression, that is, that its Expression Sequence length is 1, it is sensible to make the Number of Operands analysis a *refinement* of the Expression Sequence analysis.

Literal Value

This is a refinement of the surface type analysis. Here, for certain ‘literal’ quantities like protected symbols and integers, we attempt to track the actual concrete value of the expression (in addition to its surface type).

This degree of precision is potentially dangerous: if implemented naïvely, one might enter an expensive or infinite computation while attempting a precise estimate on some value. We must therefore must take special care in not attempting infinite or hugely expensive operations. However, since this analysis is a refinement of the Surface Type analysis, we may always fall back to merely having the surface type.

4.2.4 Summary

Summarizing, we seek to infer the following state-based properties (according to the definitions above) for a program point ℓ: the assignments that reach ℓ, the "statement contexts" that may affect ℓ. We also seek to infer the following value-based properties for a program point ℓ corresponding to a value: its surface type, its variable dependencies, its expression sequence length, its number of operands, and its literal value. Furthermore, we wish to measure the number of times that "active" variables have been written or read at point ℓ.

4.3 Modelling the properties

The properties described in Section 4.2 are of course not statically computable in general. We are therefore obliged to approximate them, and this is precisely where the apparatus of abstract interpretation will prove greatly useful.

For each property described in Section 4.2, we will provide two lattices: a concrete domain that accurately captures the quantity that we wish to learn about, and an abstract domain with which we shall approximate the property in question. As discussed in Definition 2, our lattices must satisfy the *Galois condition*:

$$L \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} M$$

where L, M are the concrete and abstract domain respectively, α, γ are the abstraction and concretization functions, respectively, and the following rule holds:

$$\alpha(X) \sqsubseteq Y \iff X \sqsubseteq \gamma(Y)$$

Some of the abstract lattices chosen are infinite and do not satisfy the Ascending Chain Condition 1. While this might give us some cause for concern, the judicious use of widening operators will ensure termination, as discussed in Section 3.6.

When we do not explicitly describe the partial order $\sqsubseteq$ on some lattice L, the partial order intended is simply the "obvious" one for this lattice. For example, given $\mathcal{P}(\{1,2,3\}) \times \mathbb{N}$, we would simply take the usual set-inclusion order $\subseteq$ on $\mathcal{P}(\{1,2,3\})$ and the $\leq$ relation on on $\mathbb{N}$, and combine these in the natural way through the Cartesian product to define the partial order on $\mathcal{P}(\{1,2,3\}) \times \mathbb{N}$.

Notation

We shall define some terms which we shall use repeatedly in the following sections.

Quantity	Description	Finite?
$\mathbf{AST}(p)$	Abstract syntax tree for program p	Yes
$\mathbf{Lab}$	Set of fresh labels (typically $\mathbb{N}$, but need not always be)	No
$\mathbf{Lab}_\star$	Finite set of labels corresponding to nodes in $\mathbf{AST}(p)$	Yes
$\mathbf{Var}$	Set of all program variables which Maple may use	No
$\mathbf{Var}_\star$	Set of program variables present in input	Yes
$\mathbf{Val}$	Set of all Maple values	No
$\mathbf{IK}$	Set of all inert tags (see 2.7) corresponding to values	Yes
$\mathbb{I}(X)$	Lattice of intervals over a totally ordered set X: $\mathbb{I}(X) = \{[a..b] : a, b \in X\}$	Finite if and only if X is finite

It it worth nothing that we have $\mathbf{Var}_\star \subseteq \mathbf{Var}$ and $\mathbf{Lab}_\star \subseteq \mathbf{Lab}$ by definition. Of course there are pratical limitations to the size of $\mathbf{Var}$, since it is defined by the behaviour of a real-world computer program, but for our purposes $\mathbf{Var}$ is unbounded.
The set $\mathbf{IK}$ is finite: in fact, its precise size is 59. For a full listing of all tags associated with values, see Appendix A.2.

Program points and annotated code

From this point forward, we will be talking frequently about *program points*; it will be helpful to clarify the meaning of this.

Given a program p, we can construct the abstract syntax tree $\mathbf{AST}(p)$ of p. We choose a set $\mathbf{Lab}$ of labels and attach a distinct label to each node of $\mathbf{AST}(p)$: we may refer to the result as an *annotated abstract syntax tree*. This permits us to define a bijection

$$\varphi : \mathbf{Lab} \to \{\text{SubTree}(x, \mathbf{AST}(p)) : x \text{ is a node of } \mathbf{AST}(p)$$

where SubTree$(x, \mathbf{AST}(p))$ is the subtree of $\mathbf{AST}(p)$ rooted at node x.

As φ is a bijection, it is reversible: note, however that merely having access to the data associated with a node does not in general allow us to identify the node or its label, as identical values may occur multiple times in an abstract syntax tree.

The reader may observe in the above that we have not said anything about the types of nodes in the AST that we are labelling: specifically, whether they are statements, expressions, or both. Many approaches to static analysis concern themselves with only one of these two (see example in [22, p. 3]). We will sometimes require both, and since we can always readily distinguish them, will include both statements and expressions in our labelling.

4.3.1 State- and program-based properties

Reaching Definitions

As this is a classical analysis, the concrete and abstract domains are well-established (see [8, 22]). Our design here follows closely the description given in [22, p. 15].

The concrete domain for this analysis can be given via a *collecting semantics* as we described in section 3.5: we have a set of *traces*, where each trace is a list of past assignments to program variables for some particular execution path of the program.

Our abstract lattice is $\mathcal{P}(\mathbf{Var}_\star \times (\mathbf{Lab}_\star \cup \{?\}))$, ordered by set inclusion. We add the extra symbol $?$ to $\mathbf{Lab}_\star$ to signify the state of a variable which has not yet been initialized. The abstraction and concretization operators are identical to those described in [22, p. 15].

For a program point p, we shall denote the reaching definitions for the *entry point* and the *exit point* of p by $\mathtt{RD}_{\text{entry}}(p)$ and $\mathtt{RD}_{\text{exit}}(p)$ respectively.

Lastly, note that we need not worry about inducing fixed points, as the Cartesian product $\mathbf{Var}_\star \times (\mathbf{Lab}_\star \cup \{?\})$ is finite.

Reaching Contexts

Reaching Contexts is a generalization of the Reaching Definitions analysis. That analysis provides invaluable information about the data flow and even control flow of a program, which may be usefully employed in other analyses. However, it is limited in some respects: the only statement type it handles specially is the assignment operator. We would like access to more statement information.

Let us define $\mathbf{Context}$ as follows:

$$\mathbf{Context} = \{\mathrm{LoopStep}, \mathrm{LoopFinal}, \mathrm{Assign}, \mathrm{If}, \mathrm{ProcedureInitialValue}\}$$

For our choice of concrete domain, we shall describe a *collecting semantics* similar to that used for Reaching Definitions above. Let us define $\mathbf{Trace}$ as follows:

$$\mathbf{Trace} = (\mathbf{Context} \times \mathbf{Var}_\star \times \mathbf{Lab}_\star)^*$$

(Here the superscript $*$ denotes the *Kleene star*.)

Each $t \in \mathbf{Trace}$ is a sequence of *contexts* from a particular trace through the procedure. Starting from the invocation of the procedure, we simply append each “context” encountered in the execution of the procedure onto a list, in exactly the same manner as a trace for Reaching Definitions maintains a list of variable assignments.

Following [22] we may define, for a trace $tr \in \mathbf{Trace}$ and $x \in \mathbf{Var}_\star$, *semantically reaching context* SRC where $\mathrm{SRC}(tr)(x)$ gives us the rightmost context in which x occurs in the trace tr.

With these definitions in place, our concrete domain is the set of program traces $\mathcal{P}(\mathbf{Trace})$, and our abstract domain is $L = \mathcal{P}(\mathbf{Context} \times \mathbf{Var}_\star \times \mathbf{Lab}_\star)$. Our abstraction and concretization functions are then simply the following:

$$\begin{array}{rcl} \alpha(X) & = & \{\mathrm{SRC}(tr)(x) : x \in \mathbf{Var}_\star \wedge tr \in X\} \\ \gamma(Y) & = & \{tr : \mathrm{SRC}(tr)(x) \in Y \text{ for all } x \in \mathbf{Var}_\star)\} \end{array}$$

Since $\mathbf{Context}$ is finite, L must also be finite and the ascending chain condition holds. Our notation is similar to that employed for Reaching Definitions. We denote the reaching contexts for a program point p by $\mathtt{RC}_{\text{entry}}p$ and $\mathtt{RC}_{\text{exit}}p$.

Number of variable reads

Let ℓ be a label corresponding to a program block. Our aim is to determine, for each active variable v, a nonnegative integer representing the number of times that v was read while control was in program block ℓ; of course for the static analysis we shall have to approximate this.

Our concrete domain is simply the formalization of our stated goal, $\mathcal{P}(\mathbf{Var}_\star \times \mathbb{N})$. Thus for each program p we have a set of pairs from $(\mathbf{Var}_\star \times \mathbb{N})$. Each element (v, n) asserts that variable v was "read" exactly n times in the context of ℓ. (This of course corresponds to a particular execution trace.)

For our abstract domain, we simply weaken our measure of the number of reads from an integer to an integer interval. Our lattice is then $L = \mathcal{P}(\mathbf{Var}_\star \times \mathbb{I}(\mathbb{N}))$: the quantity $(v, [a..b]) \in L$ expresses the fact that the number of times that v was is read in the code block ℓ was between a and b inclusive.

Note that this abstract space is infinite and does not satisfy the ascending chain condition.

Number of variable writes

This is a natural (semantic) dual to the number of reads. The domains used for its concrete and abstract semantics are identical to those used for the read-counting analysis. However, the two analyses are operationally independent.

Number of loop steps

The lattices for this property have the simplest description of the lot: given ℓ a label corresponding to a loop, we wish to measure the number of steps the loop will take. Our concrete domain is simply $\mathbb{N}$ and our abstract domain $\mathbb{I}(\mathbb{N})$.

4.3.2 Value-based properties

Surface type

In section 4.2.3 we outlined the concrete semantics of the surface type property. With each Maple value $v \in \mathbf{Val}$ we can associate a unique *inert tag* which can be computed (when v is known) by evaluating **op**(0, ToInert(v)).

For our concrete lattice take simply $\mathbf{Val}$, the set of all values. Using $\mathbf{IK}$ directly as our abstract lattice will not do; we cannot always be sure to which value a program point ℓ will correspond. Therefore we take $L = \mathcal{P}(\mathbf{IK})$ as our abstract lattice. The partial order of $\mathcal{P}(\mathbf{IK})$ is the usual subset relation.

It is straightforward to define abstraction α and concretization γ functions between the complete lattice $\langle \mathcal{P}(\mathbf{Val}), \subseteq \rangle$ of sets of Maple values $\mathbf{Val}$ and L.

Our definition means that each Maple operation mapping values to values induces, as a consequence of the the Galois connection, a natural transfer function $f : L \rightarrow L$.

As convenient as this characterization is, it is important to note that f is a coarse approximation. For example, if we encounter code resembling `a := L[1]`, we can say little about `a` because it depends on something inside `L`. Even if we knew that $\alpha(L) =$ `LIST`, the best we can do is $\alpha(a) \subseteq E$, where $E = \mathcal{P}(\mathbf{IK}) \setminus \{$`EXPSEQ`$\}$.

Expression sequence length

For our concrete domain we again take $\mathbf{Val}$. The most natural abstract lattice for expression sequence length is $\mathbb{I}(\mathbb{N})$ (the set of intervals with natural number endpoints) with $\subseteq$ given by containment.

To a program point ℓ representing a value we therefore associate a nonnegative interval $[a..b]$ with $a \in \mathbb{N}, b \in \mathbb{N} \cup \{\infty\}$.

The abstraction function α maps all values that are not expseqs to the one-point interval $[1..1]$; it maps expseq values to a range containing all possible lengths for that program point. Note that `NULL` (the empty expression sequence) maps to $[0..0]$, and that unknown expression sequences map to $[0..\infty]$.

Given a program point ℓ, we shall denote the set of expression sequence lengths of its possible values by $\mathtt{ES}(\ell)$.

The abstract lattice $\mathbb{I}(\mathbb{N})$ is illustrated in Figure 4.1, as with several other chosen abstract lattices; it is neither finite nor does it satify the Ascending Chain Condition. An example of an an infinite chain is:

$$[0..\infty] \subset [1..\infty] \subset [2..\infty] \subset \cdots$$

In Section 6.2.3 we shall see how our analysis copes with this fact.

$$\begin{tikzcd}
 & & & {[0,\infty]} \arrow[dl, dashed] \arrow[dr] & & & \\
 & & {[0,2]} \arrow[dl] \arrow[dr] & & {[1,\infty]} \arrow[dl, dashed] \arrow[dr] & & \\
 & {[0,1]} \arrow[dl] \arrow[dr] & & {[1,2]} \arrow[dl] \arrow[dr] & & {[2,\infty]} \arrow[dl, dashed] \arrow[dr, dashed] & \\
{[0,0]} \arrow[drrr] & & {[1,1]} \arrow[dr] & & {[2,2]} \arrow[dl] & & \cdots \arrow[dlll, dashed] \\
 & & & \bot & & &
\end{tikzcd}$$

Figure 4.1: The lattice $\mathbb{I}(\mathbb{N})$

4.4 Refinements

4.4.1 Refinement and Galois insertions

At this point it is necessary to interrupt the presentation of the properties and their lattices to build up some infrastructure we shall need before progressing.

As mentioned in section 4.2.3, we have designed our latter two properties, **NumOperands** and **LiteralValue**, to be *refinements* of **ExprseqLength** and **SurfaceType** respectively. By this we informally mean that **NumOperands** contains all the information present in **ExprseqLength** and some additional information not expressible in **ExprseqLength**.

Clearly this feature is both natural and highly useful. If q is a refinement of p, we may easily import information into p from q simply by "forgetting" the information specific to p, and may use p as a coarse approximation of q (as a starting point for computing q by successive refinement, for example).

Let L_p,L_q be the lattices corresponding to p,q. We should expect the refinement relationship to carry over into the chosen abstract lattices in the natural way. This is indeed the case, using an idea we have already seen: Galois connections. We should expect that if q is a refinement of p, that L_q is a concretization of L_p. That is, there should exist $\alpha_{(p,q)} : L_q \to L_p$ and $\gamma_{(p,q)} : L_p \to L_q$ satisfying the Galois criterion.

However, we want something still stronger: because q is to be a refinement of p, it must contain all the information that p does. Informally, the copy of p present in q

must be "pristine" without any spurious overapproximations. This implies that we require $\alpha_{(p,q)}(\gamma_{(p,q)}(x)) = x$ for all $x \in L_p$, and as described in Section 2 this is the condition for a *Galois insertion*.

Special lattice sum

We would like to describe the refined property as some type of lattice sum or product of the original property, in such as way as the Galois insertion is implicit in the definition. We could find no tool obviously suited to the task in the lattice theory literature (see [17, 19], for a very comprehensive list of lattice sums and products applicable to abstract interpretation, see [16]). We therefore define the following operator which will suffice.

Definition 7 *Let $\langle P, \leq, \bot_P, \top_P \rangle$ and let $\langle Q, \sqsubseteq \bot_Q, \top_Q \rangle$ be complete lattices. Let $\beta \in P$ be an* atom *of P, and $?$ is some symbol $\notin Q$. Define R as follows:*

- *If $x \in P$, then $(x, ?) \in R$.*
- *If $y \in Q \setminus \{\top_Q, \bot_Q\}$, then $(\beta, y) \in R$.*
- *Define a partial order $\preceq$ on R as follows:*
  - $(x_1, ?) \preceq (x_2, ?)$ *for all* $x_1, x_2 \in P$ *with* $x_1 \leq x_2$
  - $(\bot_P, ?) \preceq (\beta, y)$ *for all* $y \in Q \setminus \{\top_Q, \bot_Q\}$
  - $(\beta, y) \preceq (\beta, ?)$ *for all* $y \in Q \setminus \{\top_Q, \bot_Q\}$
  - $(\beta, y_1) \preceq (\beta, y_2)$ *if* $y_1 = y_2 = ?$ *or if* $y_1, y_2 \in Q \setminus \{\top_Q, \bot_Q\}$ *and* $y_1 \sqsubseteq y_2$
  - $(\beta, y) \preceq (x, ?)$ *for all* $y \in Q \setminus \{\top_Q, \bot_Q\}$ *if* $\beta \leq x$
- *Define $\bot_R = (\bot_P, ?)$ and $\top_R = (\top_P, ?)$.*

We denote R by $P \oplus_\beta Q$, and call this the "special β-sum" of P and Q.

Informally, we make a copy of the lattice P in which we replace the ideal $\{\bot_P, \beta\}$ with a copy of the lattice Q. In the new lattice, all the elements of Q sit below β but above the bottom element. (See Figure 4.2.)

This bears a resemblance to the linear lattice sum described in [17], but differs in that the "lower lattice" is inserted underneath an atom, rather than the bottom element. Notice that every element in $P \oplus_\beta Q$ is either (β, q) for $q \in Q$ or $(p, ?)$ for $p \in P$.

$$\begin{tikzcd}
 & & (\top_P, ?) \arrow[d, dashed] \arrow[ddll, dashed] \arrow[ddrr, dashed] & & \\
 & & (\beta, ?) \arrow[dl, dashed] \arrow[dr, dashed] & & \\
(p_1, ?) \arrow[drr] & (\beta, q_1) \arrow[dr] & & (\beta, q_2) \arrow[dl] & (p_2, ?) \arrow[dll] \\
 & & (\bot_P, ?) & &
\end{tikzcd}$$

Figure 4.2: *The lattice $P \oplus_\beta Q$. Here p_1, p_2, β are atoms of P and q_1, q_2 atoms of Q.*

It is clear that $P \oplus_\beta Q$ is indeed a lattice. We can show it is a complete lattice quite easily. Suppose $r_1, r_2 \in P \oplus_\beta Q$:

- If $r_1 = (\beta, y_1)$, $r_2 = (\beta, y_2)$, then $r_1 \wedge r_1 = (\beta, y_1 \sqcap y_2)$.
- If $r_1 = (x_1, ?)$, $r_2 = (x_2, ?)$, then $r_1 \wedge r_1 = (x_1 \wedge x_2, ?)$.
- Suppose $r_1 = (x, ?)$, $r_2 = (\beta, y)$. If $\beta \leq x$, then $r_1 = (\beta, y) \preceq (\beta, ?) \preceq (x, ?) = r_2$, so $r_1 \wedge r_2 = r_2$. Otherwise, $r_1 \wedge r_2 = (\bot_P, ?)$ since β is a atom.

Generalized special lattice sum

Recall that our definition of the special lattice sum $P \oplus_\beta Q$ required that the element $\beta \in P$ must be an atom. We can extend this approach to non-atoms as well.

Definition 8 *Let $\langle P, \leq, \bot_P, \top_P \rangle$ and let $\langle Q, \sqsubseteq, \bot_Q, \top_Q \rangle$ be complete lattices, and $?$ some symbol $\notin Q$.*

Let $\varphi : Q \to P$ be a monotonic function whose image $\varphi(Q)$ is an ideal of P. Then define $\langle R \preceq \rangle$ as follows:

- *If $x = \top_P$, $x = \bot_P$, or $x \in (P \setminus \varphi(Q))$, then $(x, ?) \in R$.*
- *If $y \in Q \setminus \{\top_Q, \bot_Q\}$, then $(\varphi(y), y) \in R$.*
- *Define a partial order $\preceq$ on R as follows:*
  - *$(x_1, y) \preceq (x_2, y)$ for all $(x_1, y), (x_2, y) \in R$ with $x_1 \leq x_2$ in P*

  - $(\bot_P, ?) \preceq (\varphi(y), y)$ *for all* $y \in Q \setminus \{\top_Q, \bot_Q\}$
  - $(\varphi(y), y) \preceq (\varphi(\top_Q), ?)$ *for all* $y \in Q \setminus \{\top_Q, \bot_Q\}$
  - $(\varphi(y_1), y_1) \preceq (\varphi(y_1), y_2)$ *if* $y_1 = y_2 = ?$ *or if* $y_1, y_2 \in Q \setminus \{\top_Q, \bot_Q\}$ *and* $y_1 \sqsubseteq y_2$
  - $(\varphi(y), y) \preceq (x, ?)$ *for all* $y \in Q \setminus \{\top_Q, \bot_Q\}$ *if* $\varphi(y) \leq x$

- *Define* $\bot_R = (\bot_P, ?)$ *and* $\top_R = (\top_P, ?)$.

We denote R *by* $P \oplus_\varphi Q$.

It is clear this is a generalization of the previous definition; we can recreate the previous definition with $\varphi : Q \to P$ defined as $\varphi(\bot_Q) = \bot_P$ and $\varphi(q) = \beta$ for all other values of q.

To show that $P \oplus_\varphi Q$ is a complete lattice, suppose $r_1, r_2 \in P \oplus_\varphi Q$:

- If $r_1 = (x_1, y_1)$, $r_2 = (x_1, y_2)$ with $y_1, y_2 \in \varphi(Q)$, then $r_1 \wedge r_1 = (\varphi(y_1 \sqcap y_2), y_1 \sqcap y_2)$. In fact $\varphi(y_1 \sqcap y_2) = \varphi(y_1) \wedge \varphi(y_2) = x_1 \wedge x_2$ by the definition of $P \oplus_\varphi Q$ and by the fact that φ is monotonic and $\varphi(Q)$ is an ideal.

- If $r_1 = (x_1, ?)$, $r_2 = (x_2, ?)$, then $r_1 \wedge r_1 = (x_1 \wedge x_2, ?)$.

- Suppose $r_1 = (x_1, ?)$, $r_2 = (x_2, y_2)$. If $x_2 \leq x_1$, then $r_2 = (x_2, y_2) \preceq (\varphi(\top_Q), ?) \preceq (x_1, ?) = r_1$, so $r_1 \wedge r_2 = r_2$. Otherwise, $r_1 \wedge r_2 = (\bot_P, ?)$ since $\varphi(Q)$ is a ideal.

We claim that for any φ the lattices $P \oplus_\varphi Q$ and P have a Galois insertion: the abstraction function $\alpha : (P \oplus_\varphi Q) \to P$ is defined simply as $\alpha((p, q)) = p$ for $(p, q) \in R$.

4.4.2 Additional value-based properties

We now return to cataloguing the abstract and concrete lattices.

Number of operands

Our concrete domain is **Val**. For the abstract domain, recall that our goal is to estimate the number of operands of expressions. To model this, we must first know that quantities in question *are* expressions, namely that $\mathtt{ES}(\ell) = [1..1]$. For this reason we will measure two quantities simultaneously: the expression sequence length as done in **ExprseqLength**, and also the number of operands should the quantity in question truly be an expression.

Hence we use the specialized lattice sum defined in Section 4.4.1 at the atom $[1..1] \in \mathbb{I}(\mathbb{N})$. Our abstract domain is $\mathbb{I}(\mathbb{N}) \oplus_{[1..1]} \mathbb{I}(\mathbb{N})$.

Literal Expressions

Our concrete domain is $\mathbf{Val}$. Here we want to do more than in **NumOperands** and join the two lattices at more than one point. Specifically, we define a set $\mathbf{LitVal}$ of *literal expressions*, a subset of $\mathbf{Val}$. $\mathbf{LitVal}$ consists of a number of Maple objects whose evaluated value is independent of program state and which, once evaluated, will never evaluate to anything different. $\mathbf{LitVal}$ includes rational constants, "protected" symbols such as `true`, `false`, and `FAIL`, and also strings, and lists, sets, expression sequences, and function applications made purely from objects in $\mathbf{LitVal}$.

Let α_{ST} be the abstraction function from the surface type analysis: this maps elements in $\mathbf{Val}$ to sets of inert tags. Let $\mathbf{ValTags} = \alpha_{ST}(\mathbf{LitVal})$; this is the set of all inert tags which may correspond to objects in $\mathbf{LitVal}$.

Therefore, α_{ST} is actually the map we need to define the special sum defined in 4.4.1. We therefore define our abstract domain to be $\mathcal{P}(\mathbf{IK}) \oplus_{\alpha_{ST}} \mathbb{I}(\mathbf{LitVal})$.

Though $\mathbf{LitVal}$ is a set of "simple" values, it is still infinite (the set of integers is theoretically infinite, as is the set of rationals), so we should handle termination conditions carefully.

4.5 Summary

Table 4.5 summarizes the quantities of interest. The operator names listed in the table signify the map between an arbitrary program label ℓ and the abstract space. That is, $\mathtt{STyp}(\ell)$ is the surface type abstract estimate for ℓ.

Analysis	Operator	Abstract Lattice	Finite?
ReachingDefinitions	$\mathtt{RD}_{\text{entry}}, \mathtt{RD}_{\text{exit}}$	$\mathcal{P}(\mathbf{Var}_\star \times (\mathbf{Lab}_\star \cup \{?\}))$	Yes
ReachingContexts	$\mathtt{RC}_{\text{entry}}, \mathtt{RC}_{\text{exit}}$	$\mathcal{P}(\mathbf{Context} \times \mathbf{Var}_\star \times \mathbf{Lab}_\star)$	Yes
NumReads	$\#\mathtt{R}$	$\mathcal{P}(\mathbf{Var}_\star \times \mathbb{I}(\mathbb{N}))$	No
NumWrites	$\#\mathtt{W}$	$\mathcal{P}(\mathbf{Var}_\star \times \mathbb{I}(\mathbb{N}))$	No
LoopSteps	`LSteps`	$\mathbb{I}(\mathbb{N})$	No
SurfaceType	`STyp`	$\mathcal{P}(\mathbf{IK})$	Yes
ExprseqLength	`ES`	$\mathbb{I}(\mathbb{N})$	No
NumOperands	`NOps`	$\mathbb{I}(\mathbb{N}) \oplus_{(1,1)} \mathbb{I}(\mathbb{N})$	No
LiteralValue	`LVal`	$\mathcal{P}(\mathbf{IK}) \oplus_{\alpha_{ST}} \mathbf{LitVal}$	No

Table 4.1: Abstract spaces for Properties of Interest

Chapter 5

Constraints

5.1 Introduction

The judicious application of constraints allows us to translate the semantics of Maple into a much simpler language of relational constraints which relate values in the abstract domain with variables and operators which resolve to values upon solution.

The choice of decoupling the traversal of the abstract syntax tree (the constraint generation phase) from the constraint solution phase is very important both for design purposes and comprehensibility. The constraint system for a given procedure and property presents the ultimate expression of the projection of Maple semantics into the abstract domain. In many cases, the constraint system will contain significant information which is not readily apparent in the solution, because of the need for approximation.

A good example of a simple constraint system is found in ([22] p. 8-11); this is the blueprint for our design of the constraint systems for **ReachingDefinitions** and **Reaching-Contexts**.

While Chapter 4 was concerned with a presentation of the underlying theory behind our design, this chapter will be a mixture of the specification of the constraint system and some details from the actual implementation. Here we present an overview of the constraint language for each of the properties in question

5.2 Constraints in state- and program-based analyses

For each of the properties outlined in Chapter 4, we describe those natural operations upon those properties that we employ in our analysis. We omit the description of the constraint language for **ReachingDefinitions** , as it is more or less the same as that presented in [22].

All lattices must have the usual defined lattice operations of $\sqcap$, $\sqcup$ and the constants $\top$ and $\bot$. It is important to note that these operations are (by design) forced to be lattice-specific: that is, the $\sqsubseteq$ used for **ReachingContexts** is distinct from the $\sqsubseteq$ used for **ExprseqLength** .

5.2.1 ReachingContexts

The basic operator of **ReachingContexts** is no different from that of **ReachingDefinitions**: it is a simple override operator $\bigoplus$ which transforms its argument, a set representing program state, and replaces a subset of program state with new state. For example, we have

$$\bigoplus(\{a = b, c = d, e = f\}, \{a = z, c = h\}) = \{a = z, c = h, e = f\}$$

The use of the override operator corresponds to assignment. We might observe a constraint system with entries like these:

$$\begin{gathered}
\{\mathtt{LOCAL}(2) = \{14, 27\}\} = \mathtt{RC}_{\text{entry}}(\ell_1),\\
\mathtt{RC}_{\text{entry}}(\ell_1) = \mathtt{RC}_{\text{exit}}(\ell_1),\\
\mathtt{RC}_{\text{exit}}(\ell_1) \sqsubseteq \mathtt{RC}_{\text{entry}}(\ell_2), \mathtt{RC}_{\text{entry}}(\ell_2) = \mathtt{RC}_{\text{exit}}(\ell_2)\\
\mathtt{RC}_{\text{exit}}(\ell_2) \sqsubseteq \bigoplus(\mathtt{RC}_{\text{entry}}(\ell_3), \{\mathtt{LOCAL}(1) = \{\ell_2\}\})\\
\mathtt{RC}_{\text{entry}}(\ell_3) \sqsubseteq \mathtt{RC}_{\text{exit}}(\ell_4),\\
\mathtt{RC}_{\text{exit}}(\ell_4) \sqsubseteq \mathtt{RC}_{\text{entry}}(\ell_5)
\end{gathered}$$

What is significant about **ReachingContexts** and the reason why it we widely rely upon it as an enabler for other analyses, lies not in the operator chosen but in the nature of the data being overridden. By the description mentioned in Section 4.3.1, the state in **ReachingContexts** includes information about the current context of the program point, which includes information about loop control and conditional structures in addition to the assignments. The result is an augmented variant of **ReachingDefinitions** which performs

```
p := proc(N::integer ,a,b) local n, i, s
     i := 1;
     [n := f(a,b);]$^{\text{before\_state}}$
     while [n > 0]$^{\text{cond\_state}}$ then
          [n := n − N;
              i := i + 1]$^{\text{loop\_state}}$;
     end if;
     [print("Result is ", n);]$^{\text{after\_state}}$
end proc:
```

Figure 5.1: **ReachingContexts** Example

control flow as well as data flow analysis. Figure 5.1 provides an example. The annotations in the body of p in Figure 5.1, labelled from 1 to 4, correspond to the state before the loop, the state at the execution of the loop condition, the state after the execution of the loop body, and the state after the loop termination, respectively.

Let Δ represent the set of variable transformed by the loop at code point ℓ (in this case, it is the two local variables n and i. We then introduce the following symbolic quantities for each variable $v \in \Delta$ altered in the loop:

- $\texttt{LoopStepInit}(v, \ell)$ - the initial value of variable v within a loop step
- $\texttt{LoopFinal}(v, \ell)$ - the final value of variable v after the loop exits

$$iv = \{v = \texttt{LoopStepInit}(v, \ell), v \in \Delta\} \quad fv = \{v = \texttt{LoopFinal}(v, \ell), v \in \Delta\}$$

Our aim is to introduce these symbolic quantities into our relational system for **ReachingContexts**. The classical, straightforward formulation of **ReachingDefinitions** will contain the following:

$$\begin{array}{llll} \texttt{RD}_{\text{entry}}(\text{cond}) & \sqsubseteq \ \texttt{RD}_{\text{exit}}(\text{before\_state}), & \texttt{RD}_{\text{entry}}(\text{cond}) & \sqsubseteq \ \texttt{RD}_{\text{exit}}(\text{loop\_state}), \\ \texttt{RD}_{\text{entry}}(\text{after\_state}) & \sqsubseteq \ \texttt{RD}_{\text{exit}}(\text{cond}), & \texttt{RD}_{\text{entry}}(\text{after\_state}) & \sqsubseteq \ \texttt{RD}_{\text{exit}}(\text{loop\_state}) \end{array}$$

These merely express the normal statement semantics: evaluation of cond may occur either following the preceding statement (when the loop begins) or after a loop iteration; evaluation of after_state occurs after the loop aborts, either after the condition is checked or after the statement sequence is executed.

In **ReachingContexts**, these relations are not generated. Instead, given this input, we would generate the following:

$$\begin{array}{lcl}
\mathtt{RC}_{\text{entry}}(\text{cond}) & \sqsubseteq & \bigoplus(\mathtt{RC}_{\text{exit}}(\text{before\_stat}), iv) \\
\mathtt{RC}_{\text{entry}}(\text{cond}) & \sqsubseteq & \bigoplus(\mathtt{RC}_{\text{exit}}(\text{loop\_stat}), iv) \\
\mathtt{RC}_{\text{entry}}(\text{after\_stat}) & \sqsubseteq & \bigoplus(\mathtt{RC}_{\text{exit}}(\text{cond}), fv) \\
\mathtt{RC}_{\text{entry}}(\text{after\_stat}) & \sqsubseteq & \bigoplus(\mathtt{RC}_{\text{exit}}(\text{loop\_stat}), fv)
\end{array}$$

The effect of this is that all loop variables are replaced with a *symbolic dummy variable*, which indicates the state of the variable is controlled by the loop. Note this happens not merely with the loop control variable (if there is one) but with every variable which is altered by the loop.

The effect is to completely partition the loop state from the state of the rest of the program. After the rest of the program has been solved or simplified, the loop can be analyzed independently and the results substituted into the solution. This general approach of isolating and reserving pieces of program state for special treatment is known as *partitioned iteration* (see [7]). Note that variables appearing only in a read context will not be overridden thus; however, since they appear only in a read context it is completely safe to leave them untouched.

The idea of replacing an unknown state with a symbolic one within the constraint language is one we shall use several times over.

5.2.2 NumReads and NumWrites

The constraint language here is quite simple. Aside from meet and join there are only two operations on values: addition and scalar multiplication. All other values are either type variables $\mathbf{NumReads}(\ell)$, $\mathbf{NumWrites}(\ell)$ or the base, which is a set of ordered pairs $(v, \mathfrak{I})$, where v is a program variable and $\mathfrak{I}$ records an estimate on how many times it has been read or written.

Operations on the sets are passed elementwise onto the contents:

$$\begin{array}{lcc}
\sum\left(\{v = [a..b], w = [e..f]\}, \{v = [c..d]\}\right) & = & \{v = [a + c..b + d], w = [e..f])\} \\
\bigsqcap(\{v = [2..4], w = [0..\infty]\}, \{v = [3..5]\}) & = & \{v = [3..4]\} \\
\bigsqcup(\{v = [2..4], w = [0..\infty]\}, \{v = [3..5]\}) & = & \{v = [2..5], w = [0..\infty]\}
\end{array}$$

5.3 Constraints in value-based analyses

The constraint languages for the four valued-based properties (**SurfaceType**, **ExprseqLength**, **NumberOfOperands**, and **LiteralValue**) are broadly similar. For brevity's sake we shall use **ExprseqLength** as an example throughout which is characteristic of the lot of them. In addition to the base types (e.g. $\mathbb{I}(\mathbb{N})$) and the constraint variables corresponding to types, there are several other features to the constraint language. Most important are *lifted operators*. These correspond to operators or functions in the concrete interpretation (i.e. in Maple) which have been "lifted" via the Galois connection to functions $\phi : X^n \to X$, where X is the abstract domain.

5.3.1 ExprseqLength

One such is example for **ExprseqLength** is *expseq* operator (sometimes called the comma operator), which can be viewed as taking n expression sequences and appending them to produce a single one.

```
> (1,2,3), (4,5,6);
                                            1, 2, 3, 4, 5, 6
```

Because the **ExprseqLength** domain is defined as a measure of expression size, in this domain the lifted expseq function actually behaves as an addition operator on the base type of intervals

$$\mathbf{LiftFunction}(\mathtt{EXPSEQ})([1..4],[2..3]) = [3..7]$$

Another important function is the *evalb operator*. Because Maple implicitly uses different semantics when evaluating conditional expressions in a loop or if condition, it is necessary to take this into account and build a special function that accounts for the special semantics. The evalb operator will force a true/false result from conditional expressions when one would otherwise not be obtained; this can affect the size or type of the results.

5.3.2 NumOperands

As the base type for **NumOperands** is an ordered pair, some of the operations need special care for definition. This is the example of $\mathbf{LiftFunction}(\texttt{EXPSEQ})$ used above:

$$\begin{aligned}
\mathbf{LiftFunction}(\texttt{EXPSEQ})(([a..b]),([c..d])) &= ([a+c..b+d]) \\
\mathbf{LiftFunction}(\texttt{EXPSEQ})(([1..1],\mathfrak{I}),([1..1],\mathfrak{I})) &= ([2..2])
\end{aligned}$$

In **NumOperands** we also define two new operators to deal specifically with the fact that the base type is a peculiar kind of cartesian product. We have two primitives for moving range estimates from one element to the other:

$$\begin{aligned}
\texttt{Wrap}(([a..b],[c..d])) &= ([1..1],[a..b]) \\
\texttt{Flatten}(([a..b],[c..d])) &= \begin{cases} ([c..d],?) & \text{if } a=1, b=1 \\ \perp & \text{otherwise} \end{cases}
\end{aligned}$$

The names are inspired by Maple semantics. An expression sequence is “wrapped” inside a list, while a list may optionally be “flattened” to produce an expression sequence. As we might expect, they act as inverses:

$$\begin{aligned}
\texttt{Wrap}(\texttt{Flatten}(\mathfrak{a})) &= \mathfrak{a} && \text{whenever } \texttt{Flatten}(\mathfrak{a}) \text{ is defined} \\
\texttt{Flatten}(\texttt{Wrap}(([a..b],[c..d]))) &= ([a..b],?) && \text{for } [a..b],[c..d] \in \mathbb{I}(\mathbb{N})
\end{aligned}$$

Chapter 6

Design

While the vocabulary we have constructed for capturing the semantics of Maple in our chosen properties is highly expressive, we have as yet not said much with it. Here we will discuss details involved in the generation of the constraint systems described in Chapter 5 and describe our approaches for the solutions of these systems. These include the use of widening operators and a limited form of solution of loop constraints through the use of recurrence relations.

Finally, we will provide a sketch of the specification for the software system written in Maple to perform these analyses.

6.1 Constraint generation

Ultimately, all constraints are generated through analysis of the abstract syntax tree for an input procedure p. State- and program-based analyses simply traverse the abstract syntax tree and build up a large conjunction of constraints directly as a result of this traversal.

In constrast, the constraints generated for a value-based property are divided into two classes: *opportunistic constraints* and constraints generated from **ReachingContexts** results.

6.1.1 Opportunistic constraint generation

A so-called "opportunistic" constraint rule is a rule which attempts to match some part of the input during AST traversal against some particular pre-coded pattern. In the event a pattern is matched, an constraint is generated and added to the system.

Opportunistic rules are the usual method of constraint generation for all state- and program-based analyses, and are also used widely in value-based properties. Additionally, value-based analyses utilize other generated constraints (generated from **ReachingContexts**, a state-based analysis) in order to save us some work and rely on already-computed information.

As said earlier, opportunistic constraints are generated directly from a pass through the AST. Each analysis has a custom ruleset that tests for certain patterns in the AST. For example, if an integer constant is seen at point ℓ the generator for **ExprseqLength** will produce $\{\mathtt{ES}(\ell) \sqsubseteq [1..1]\}$, since an integer must always have expseq size 1.

While other generated constraints provide the glue that properly implements the semantics of the Maple language, these opportunistic rules provide the raw material that give different analyses their unique colour. The design of these rulesets is purely *ad hoc*, and the rule designer must be very familiar with the semantics of the language in question: it is exceedingly easy to miss some boundary case in the formulation of a rule. Even the reader familiar with the language in question may be surprised by what conclusions may be drawn from certain code patterns.

Several examples of “opportunistic” rules used by the system are given in Appendix C.

LoopSteps

At this point is it worth revealing that one of our chosen properties, **LoopSteps**, is essentially a “cheat.” The only effort **LoopSteps** itself ever makes to generate constraints restricting the range of possible loop steps is a trivial case when there is a for-loop with no condition and purely numeric starting, terminating, and increment values. In no other circumstances does **LoopSteps** ever generate by itself any constraints.

Why then do we have **LoopSteps**? The answer is that because our inferred properties trade data with each other after every iterate, **LoopSteps** may “learn” loop-step information from someone else. In particular

- For for-in loops which traverse data structures, **ExprseqLength** or **NumOperands** may provide the size of the data structure in question
- **LiteralValue** may be able to conclude that the continuation condition for the loop is exactly “true” or “false” under certain circumstances.

The idea then is that **LoopSteps** is a convenient dropping-off point for information any other analysis may happen to have about loop size.

6.1.2 Generation of constraints from Reaching Contexts

There are clearly limits to what can be obtained from "opportunistic constraints" alone: in solving constraint systems for value-based properties, we need a way to unite our approximations for variables which have the same assigned value but occur at different program points. We achieve this using the information from Reaching Contexts as decribed earlier.

The main idea is that, given any two program points ℓ_1, ℓ_2, we can compute a constraint set $f(\ell_1, \ell_2)$ as follows:

1. Check if ℓ_1 and ℓ_2 are both instances of the same program variable, say v. If not or if $\ell_1 = \ell_2$, then set $f(\ell_1, \ell_2) = \emptyset$ and exit.
2. Otherwise, check $\texttt{RC}_{\text{exit}}(\ell_1)$ and $\texttt{RC}_{\text{exit}}(\ell_2)$ and look up the reaching contexts of variable v in each. Let $S_1 = \texttt{RC}_{\text{exit}}(\ell_1)\langle v\rangle$ and $S_2 = \texttt{RC}_{\text{exit}}(\ell_2)\langle v\rangle$.
3. Then define

$$f(\ell_1, \ell_2) = \begin{cases} \{\text{ValProp}(\ell_1) \sqsubseteq \text{ValProp}(\ell_2)\} & \text{if } S_1 \subseteq S_2 \\ \{\text{ValProp}(\ell_1) = \text{ValProp}(\ell_2)\} & \text{if } S_1 = S_2 \\ \{\text{ValProp}(\ell_1) \sqsupseteq \text{ValProp}(\ell_2)\} & \text{if } S_1 \supseteq S_2 \\ \emptyset & \text{otherwise} \end{cases}$$

   (where $\text{ValProp} \in \{\texttt{ES}, \texttt{STyp}, \texttt{NOps}, \texttt{LVal}\}$)

We therefore simply compute $C = \cup_{x,y\in L} f(x, y)$ and augment the set of "opportunistic" constraints generated with the newly computed set C.

The central idea here is that the relationship we indicate between the constraint variables corresponds to the shared past history of the underlying program variables.

Note that this approach might lead to overapproximation. Consider the following:

```
ListOrIntegerToSet := proc(x)
    if type(x, integer) then
        {x}
    elif type(x, list) then
```

```
        convert(x, set)
    end if;
end proc:
```

In this example the variable N is never assigned, therefore its past computation history is the same everywhere. However, in practice there is no possible situation under which both one execution path could pass through both program points (as nothing can be both a list and an integer simultaneously.) This does not affect the correctness of our result, only its precision, as we merely overapproximate.

6.1.3 Generating constraints for “unpartitioning”

Variables in the constraint language for value-based properties come in two varieties. The first are the typical kind: these associate a program point ℓ representing a Maple value with a value in the abstract domain.

The other sort are “control variables”: they associate a program variable and some control structure in the AST with a value in the abstract domain. There are three sorts of these these variables, corresponding to loops, if-then-else conditions, and procedures respectively.

Only the variables of the first kind are “native” to this analysis in the sense that they are derived directly from a traversal of the AST that is specific to this analysis. We are capable of performing the analysis without control variables, but the use of control variables allows us to better capture the special semantics of various control structures.

The control variables are generated from data in **ReachingContexts**, and are introduced into the constraint system by by equating them to combinations of existing variables. We illustrate the process here for loops; there is an analogous process for conditional structures and procedure state.

Let ℓ be some loop and Δ be the set of program variable transformed by the loop. For some value-based property $\mathrm{VP} \in \{\mathtt{ES}, \mathtt{STyp}, \mathtt{NOps}, \mathtt{LVal}\}$ and each $v \in \Delta$, we define the following:

Quantity	Definition	Explanation
$\texttt{LoopInit}(v, \ell)$	$\bigsqcup(\mathrm{VP}(w) : w \in \texttt{RC}_{\text{exit}}(a, v))$ where a is the label of the last statement before the loop.	initial value of v when the loop begins.
$\texttt{LoopFinal}(v, \ell)$	$\bigsqcup(\mathrm{VP}(w) : w \in \texttt{RC}_{\text{exit}}(b, v))$ where b is the label of the loop block.	the final value of v when the loop terminates (note we do not implicitly assume termination)
$\texttt{LoopStepInit}(v, \ell)$	$\bigsqcup(\mathrm{VP}(w) : w \in \texttt{RC}_{\text{entry}}(c, v))$ where c is the label of the conditional block.	value of v at the start of a loop step (unlike $\texttt{LoopInit}(v, \ell)$, this may follow an indeterminate number of previous steps).
$\texttt{LoopStepFinal}(v, \ell)$	$\bigsqcup(\mathrm{VP}(w) : w \in \texttt{RC}_{\text{entry}} d, v)$ where d is the label of the loop statement block	value of v at the end of a loop step (analogously to $\texttt{LoopStepInit}(v, \ell)$).

It is important to make a distinction between initial/final states of the loop overall, and initial/final states of the looping block.

As described earlier for Reaching Contexts, the purpose of the introduction of these variables is to partition program states in such a way as to make it easier to refine results. Specifically for loops, we isolate the loop state from the state of the rest of the program, after which we may attempt to solve the loop iteratively (see [7]).

Clearly we do not want the states permanently partitioned. For assignments and if structures, we add information back in a controlled manner, imposing certain relations between generated variables specific to a particular control structure.

6.2 Constraint solution

Our constaint system is always a simple disjunction of relations. We therefore have no need to worry about disjunctions, or conversion to normal form. The following is a brief outline of our method of solving constraint systems.

6.2.1 Outline

The process for constraint solution looks something like this. Because of the duality principle of lattices, we can freely replace all lattice terminology with ther dual (e.g. replace $\top$ with $\bot$, $\sqsubseteq$ with $\sqsupseteq$, etc.) without changing the conclusions of the sentence.

1. Gather all equality relations and assemble all constraints into equivalence classes based on equality constraints. From this point on we need no longer worry about equality constraints at all: they have been dealt with.

2. We now collect all strict relations ordered by their rightmost quantity. By traversing all the relational constraints, assemble the sets

$$SR(y) = \{x : x \sqsubseteq y \text{ is a constraint in the system } \}$$

   This is in practice equivalent to declaring that the value of y will be approximated by $\bigsqcap SR(y)$; this idea may be viewed as a variant of the "no junk rule" which holds that y should at most be the join of all quantities "under" it; if nothing is constraining y to be larger, it should not be larger.

3. Initialize all constraint variables to $\top$ (the lattice top). Denote the current approximation of a constraint variable by $A[x]$.

4. Repeatedly traverse every y for which $SR(y)$ is defined. For each $x \in SR(y)$:

   (a) If x is not a value or variable, but is an operator $f(z_1, \ldots, z_n)$, evaluate $r := f(A[z_1], \ldots, A[z_n])$, where the $A[z_i]$ are the latest approximation for z_i. Assign $A[x] := A[x] \sqcap r$.

   (b) Set $A[y] := A[y] \sqcap A[x]$.

This approach will ultimately succeed as long as there are no infinite chains in our analysis. It is essentially a simple form of Chaotic Iteration, in which we exert very little effort to optimize the speed of our solution.

6.2.2 Flow of information

It is worthwhile to observe that, generally speaking, flow of information in our abstract analysis will follow the flow of the (concrete) program.

The reason for this should become clear upon examining the graph on constraint variables which is induced by the partial order between them imposed by constraint relations. Because the relations are induced by shared past computation history, we have $a \sqsubseteq b$ if constraint variable a has a subset of the past computation history of variable b. This frequently means that b is "from the future", and has since acquired additional computation history (e.g. through unification of conditional branches.)

Similarly, in evaluated constraints in which one of the constraint values is not a base type or variable, but a constraint operator, the pattern is typically $x_{\text{result}} \sqsubseteq f(x_1, \ldots, x_n)$ for some variables $x_1, \ldots, x_n$. This type of operator effectively behaves as the direct analogue of a concrete function (indeed, it often *is* a lifted function) and does not pass information into its arguments $x_1, \ldots, x_n$ but only its result x_{result}.

Backward-propagating constraints

The exceptions to this rule of forward propagation are interesting enough to warrant mention. The first example is a subset of lifted library functions whose constraints take the form

$$x_{\text{result}} \sqsubseteq \mathbf{LiftFunction}(\texttt{FUNCTION}, fname)(x_1, \ldots, x_n)$$

Because these functions type-check their arguments before proceeding, our Function Property Table reflects this fact, and the $\mathbf{LiftFunction}$ call generates a series of constraints $x_i \sqsubseteq T_i$ which correspond to restrictions imposed on the range of inputs to $fname$ as a result of type-checking. Nevertheless, this backwards propagation is highly limited as the T_i are always constants, and does not constitute a series counterexample to this trend..

A slightly more convincing example is used in **ExprseqLength** in handling $\mathbf{LiftFunction}(\texttt{EXPSEQ})$. As was mentioned in Chapter 5, this operator acts like a sum on $\mathbb{I}(\mathbb{N})$. We therefore have a special case that recognizes the pattern

$$x_{\text{result}} = \mathbf{LiftFunction}(\texttt{EXPSEQ})(x_1, \ldots, x_n)$$

If, during the solution phase, the approximation $A[x_{\text{result}}]$ is known to be finite, we actually do *interval subtraction*, computing for each i the quantity

$$r_i = x_i - \sum_{j=1}^{i-1} - \sum_{j=i+1}^{N}$$

where the addition and subtraction here are *interval* addition and subtraction, that is:

$$[a..b] - [c..d] = [\max(a-d,0)..\max(b-c,0)]$$

This extra step will never help in the refinement of the result x_{result}, but may improve the precision of the x_is.

For an exposition on a type of this problem, restricted to the two-variable case, see the thesis of Antoine Miné [21] .

6.2.3 Termination conditions

The issue of ensuring termination is one that cannot be avoided. By our previous arguments about the analysis mirroring control flow, it should be apparent that because the only Maple constructs that allow for the repeated visitation of a single program point are recursive calls and loops. There are therefore our main concerns as far as termination is concerned.

In a procedure with no loops or recursive calls, we can generally say that all the constraints point in the same direction: the constraint graph is a *directed acyclic graph*, can therefore never loop, so termination is guaranteed because the code is finite.

6.2.4 Loops

The proper handling of program loops is a major component of our analysis. We have two distinct strategies for dealing with loops. Both rely on the our ability to recognize constraint variables used in a loop context using the loop constraint variables described in Section 6.1.3 and generated by the use of Reaching Contexts.

As described earlier, in all value-based analyses, for every variable v transformed by a loop ℓ we have introduced into the language of constraints the four quantities `LoopInit`(v,ℓ), `LoopFinal`(v,ℓ), `LoopStepInit`(v,ℓ), and `LoopStepFinal`(v,ℓ) in order to partition the constraint graph of the program to prevent cycles.

However, this gives us a convenient means of expressing the problem as a *recurrence relation*. The constraint systems will induce a symbolic dependency of the final state of a loop iteration upon the initial state of the iteration. In other words, for a loop ℓ with loop variables $v_1, \ldots, v_n$ we have the following:

$$\begin{array}{rcl} \texttt{LoopStepFinal}(\ell, v_1) & \sqsubseteq & F_1(\texttt{LoopStepInit}(\ell, v_1), \ldots, \texttt{LoopStepInit}(\ell, v_n)) \\ \vdots & & \vdots \\ \texttt{LoopStepFinal}(\ell, v_n) & \sqsubseteq & F_n(\texttt{LoopStepInit}(\ell, v_1), \ldots, \texttt{LoopStepInit}(\ell, v_n)) \end{array}$$

or, in matrix form

$$\texttt{LoopStepFinal}(\ell, \check{v}) \sqsubseteq F(\texttt{LoopStepInit}(\ell, \check{v}))$$

Since $\texttt{LoopStepInit}(\ell, \check{v}) = \texttt{LoopInit}(\ell, \check{v})$ at the beginning, we can therefore simulate the effect of the loop simply by computing the following, where n is the number of loop steps:

$$\texttt{LoopStepFinal}(\ell, \check{v}) \sqsubseteq F^n(\texttt{LoopInit}(\ell, \check{v}))$$

We have two approaches for performing this computation: a *partitioned iteration approach* and a *recurrence relation approach*.

Partitioned iteration approach

The most obvious, most readily used, and most general solution is simply to simulate the execution of the loop. Using the loop variables introduced in Section 6.1.3, we construct a mini-procedure consisting only of the constraint variables that are dependent on the loop.

We have some kind of estimate from $\mathbf{LoopSteps}$ on the number of steps a given loop ℓ will take; this may well be $[0..\infty]$, which is no great loss. As our goal we simply iterate $X, F(X), F^2(X), \ldots$ repeatedly until we have hit a fixedpoint or taken more than $\max(\texttt{LSteps}(\ell)))$ steps.

We are guaranteed to converge provided we employ widening operators (see 3.6.1) in the case of analyses on infinite lattices; fortunately, we indeed use widening operators in our analyses for **ExprseqLength** and **NumberOfOperands**, both infinite lattices.

The particular operator chosen is a variant, adapted for $\mathbb{I}(\mathbb{N})$ of the operator ∇_K described in [22] (p. 226-227). Essentially, the principle is that in cases of uncontrolled growth, the upper bound ∇_K is widened to the nearest element in a finite set K of integers; if all elements of K have been exceeded, ∞ is returned. The finiteness of K ensures no infinite chains are possible.

We do not currently employ narrowing operators to refine our widened results, though this would not be difficult to implement.

We return the result to the calling procedure, substitute the result for our symbolic quantities `LoopInit`,`LoopFinal`,`LoopStepInit`,`LoopStepFinal` into our solution, and we are done.

6.2.5 Recurrence relation approach

An alternate approach investigated and implemented was that of treating the loop as a *recurrence equation*. This approach has great utility, chiefly because it does not involve a potentially expensive simulation of the loop execution within the abstract domain, as the partitioned iteration approach does. The chief objection to this approach is that number of cases it can handle are quite small.

Generalized characteristic function

We begin our discussion with a simple analysis to handle the case of whether the loop executes at all.

Let ℓ be the label of some loop and v be a variable whose state is transformed by ℓ. Suppose α is our best approximation of v just before the loop begins, and β is our approximation at the end of the loop body, after one or more passes through the loop.
Suppose ℓ executes n times. We can formalize the value of v immediately after the loop block by the following:

$$\chi_{\mathbb{N}}(n,\alpha,\beta) = \begin{cases} \alpha & \text{if } n = 0 \\ \beta & \text{if } n \geq 1 \end{cases}$$

Observe that $\chi_{\mathbb{N}}$ is essentially a simple characteristic function.
However, in static analysis we usually have only an approximation of the value of n. Let $\mathfrak{I}$ be such an interval estimate. Then define the *generalized characteristic function*

$\chi_{\mathbf{I}(\mathbb{N})}(\mathfrak{I}, \alpha, \beta)$ by

$$\chi_{\mathbf{I}(\mathbb{N})}(\mathfrak{I}, \alpha, \beta) = \begin{cases} \alpha & \text{if } \mathfrak{I} = [0..0] \\ \beta & \text{if } \mathfrak{I} \sqsubseteq [1..\infty] \quad \text{(i.e. } \mathfrak{I} = [a..b] \text{ and } a \geq 1\text{)} \\ \alpha \sqcup \beta & \text{otherwise} \end{cases}$$

Notice that $\chi_{\mathbf{I}(\mathbb{N})}$ has the useful property:

$$(\alpha \sqcap \beta) \sqsubseteq \chi_{\mathbf{I}(\mathbb{N})}(\mathfrak{I}, \alpha, \beta) \sqsubseteq (\alpha \sqcup \beta)$$

The name is motivated by the fact that this is the best-possible generalization of our original $\chi_{\mathbb{N}}$ to the interval lattice $\mathbf{I}(\mathbb{N})$.

Recurrence relations

As above we have

$$\texttt{LoopStepFinal}(\ell, \check{v}) \sqsubseteq F(\texttt{LoopStepInit}(\ell, \check{v}))$$

This can be viewed as a recurrence relation, where $\texttt{LoopStepFinal}(\ell, \check{v})$ is the $i + 1$st value, $\texttt{LoopStepInit}(\ell, \check{v})$ the ith, and $\texttt{LoopStepInit}(\ell, \check{v})$ the initial condition.

For certain values of F, we can solve this recurrence explicitly. In particular, if we restrict ourselves to the case when F is *diagonal*, we need only worry about one loop variable v_i at a time.

We have compiled a rather *ad hoc* set of patterns for diagonal functions F which we can handle. We will use one, from **ExprseqLength**, as an illustrative example:

$$\texttt{LoopStepFinal}(\ell, v_i) \sqsubseteq \textbf{LiftFunction}(\texttt{EXPSEQ})(\texttt{LoopInit}(\ell, v_i), c)$$

for some constant c. As was mentioned in Chapter 5, this lifted function acts an addition operator, so if we knew n we could express a solution simply as

$$\texttt{LoopStepFinal}(\ell, v_i) = \texttt{LoopInit}(\ell, v_i) + n \cdot c$$

where $+$ is interval addition and $\cdot$ represents interval scalar multiplication.

Of course, in general we only ever have an interval bound $\texttt{LSteps}(\ell)$ on n. In this case, we can express the explicit solution to the recurrence as

$$\texttt{LoopStepFinal}(\ell, v_i) = \texttt{LoopInit}(\ell, v_i) + \texttt{LSteps}(\ell) * c$$

where all the quantities on the right-hand side have been approximated, and $*$ is interval multiplication (as distinct from interval *scalar* multiplication.).

In general, when we can solve the recurrence we will obtain an equation $\texttt{LoopStepFinal}(\ell, v_i) = d$ which, for convenience, assumes the loop has executed at least once. We can then solve $\texttt{LoopFinal}(\ell, v_i)$ using $\chi_{\mathbf{I}(\mathbb{N})}$ as defined earlier, by assigning

$$\texttt{LoopFinal}(\ell, v_i) = \chi_{\mathbf{I}(\mathbb{N})}(\texttt{LSteps}(\ell), \texttt{LoopInit}(\ell, v_i), d)$$

This approach conveniently skips the simulation of the loop, and furthermore avoids any imprecision enforced by the use of widening operators. Nevertheless it is very difficult to anticipate the types of recurrences that actually come up in practice, many of whose transfer functions are lifted library functions about which we cannot easily reason. The recurrence relation approach is highly worthwhile when a solution is possible.

6.2.6 Function application

Function applications pose several problems for our analysis: in understanding what information is being passed in to the function application, making as precise as possible the information that is being passed out, and in handling termination issues.

Digesting function arguments

The most immediate problem is finding out how many arguments are being supplied to the function and what program values they correspond to. This problem probably does not occur to one unfamiliar with Maple or other languages (this includes Perl) that support the equivalent of expression sequences.

If we encounter `f(a,b)` we must first, before anything else, establish whether `a` and `b` are expression sequences before concluding `f` is being called with two arguments. Furthermore, even knowing that `f` is called with two arguments does not uniquely identify `a` and `b` as the arguments: there are the two spurious cases in which one of the two variables has expseq length of 2 and the other equal to `NULL` (the empty expression sequence).

One of our analyses, **ExprseqLength**, is designed to accomplish this task; this is not accidental. Nevertheless, even with **ExprseqLength** there will be many cases in which we cannot uniquely match values in the function application with procedure parameters, because not enough is known about the arguments provided or the number of parameters `f` will accept.

Currently, we only handle procedure arguments when we can exactly match them to parameters. This could be generalized to cases where there are a small number of possible matchings (such the example above with `f`, `a`, and `b`).

Handling built-in functions and special names

As discussed in Chapter 2, many procedures in Maple are not writtenin the Maple language, but are built into the Maple kernel and usable transparently by programs.

These procedures are entirely outside our reach for analysis, and because they make up much of the core functionality upon which many programs depend, we shall need our analyses to recognize and know something about them.

To reconcile these problems, we have built a *database of builtin functions* and another for *special names*, which are symbols with special significance to the system (examples include `true`, `false`, and `Pi`.)

In these databases, we have encoded by hand the results which our analyses should have found, had they been able to examine the procedures in question. The properties encoded in the table were gathered through a combination of reading the formal specification of the builtin procedure (from the online help system or [18]) and direct experimentation upon the builtin procedures within a session.

Handling previously-analyzed procedures

So far we have spoken purely of analyzing single procedures, and said little about function applications except for standard library functions. Yet it is frequently the case that we wish to analyze large libraries of heavily interdependent code as a whole.

Moreover we would hope that if we have analyzed g and are now analyzing f which calls upon g, we should be able not only to benefit from our previous analysis but would hope to *specialize* it to the context in which g is used within f.

We have implemented two approaches to dealing with function calls to previously-analyzed procedures. One is purely superficial. The other is more sophisticated but also more costly.

Suppose in the context of executing f we encounter a function call to g, which has been analyzed before with constraint set C_g and solution S_g. (We also assume we can properly match the parameters of g!) Suppose the function call for g within f has n arguments, which in the language of the abstract domain are $x_1, \ldots, x_n$.

Superficial approach: Let $p_1, \ldots, p_n$ be the constraint variables corresponding to the parameters of g in the abstract domain, and let r be the constraint variable corresponding to its return value in the abstract domain. The current estimates $S_g(p_1), \ldots, S_g(p_n), S_g(r)$ were computed when g was previously analyzed. We generate some constraints from the knowledge of the parameters and return value of g:

$$\Gamma_1 = \left((g(x_1, \ldots, x_n) \sqsubseteq S_g(r)) \wedge \bigwedge_{i=1..N} (x_i \sqsubseteq S_g(p_i)) \right)$$

We augment our constraint system in f with Γ_1, and move on. We do not re-execute g or examine it further at all.

Specialization approach: In the specialization approach, we make a temporary copy of the constraints and solution for (C_g, S_g); call this copy (C'_g, S'_g). We define a constraint system which looks very much like the previous, but with all relations reversed:

$$\Gamma_2 = \left(r \sqsubseteq S_f(g(x_1, \ldots, x_n)) \wedge \bigwedge_{i=1..N} (p_i \sqsubseteq S_f(x_i)) \right)$$

Here S_f is the current solution to the outer function f: since we are in the middle of solving it, there must at least be *some* solution for f in existence.

We then solve $C'_g \wedge \Gamma_2$ and assign the result to S'_g. We have created and solved a *specialized* version of g, which we incorporate into our constraint system for f in the same manner as before. This specialization is somewhat akin to *partial evaluation* within the constraint language. Recursive calls present no immediate problem: as with other procedures, we simply use the most recent complete solution.

While the second approach clearly gives better results, there is an issue of where to stop. We have no immediate equivalent of *dead-code elimination*: if the list of dependent procedures on f is large, this could be a costly or infinite computation.

While this is probably best handled by a theoretically sound approach like narrowing and widening, in practice our implementation simply has a preset depth threshold (default value 1). After specializing every called procedure to a depth of d, we simply use the superficial results described earlier and return. With this threshold in place we need not worry about convergence, since the number of called procedures must be finite.

6.3 Software specification

We wish to design a piece of software which accepts as input a *procedure* p, constructs a *property record* for p, and saves the record for p to a *property database* from which it may later be retrieved without the need for recomputation.

The property record for p is all the collected and inferred information we have about p, including its abstract syntax tree, any generated constraints, and any solved properties.

6.3.1 Scope

We have stated that we wish to analyze Maple procedures. We will attach one important proviso to this claim which slightly restricts the generality of our results. We will not handle or support the `try/catch/finally` constraint in Maple.

Because the try/catch system can involve the immediate transportation from almost any program point to one of several “catching” clauses, supporting this construct would have serious consequences for our design. Furthermore we would be obligated to abandon our central assumption that the source code in its present format was expected not to throw an error.

We do treat procedures with try/catch/finally structures in them, but entirely ignore the data outside the `try` block and issue a warning when first encountering such a procedure.

6.3.2 Preprocessing

Prior to any real work, we perform some preprocessing work to gather some basic data about the input. These steps include the following:

- A pass to determine whether loops have nontrivial conditions, or are pure for loops.
- A pass to record whether each label is inside an "evalb context". This is a special context which Maple implicitly imposes when evaluating conditional expressions; see section 5.3.1.
- A pass to determine whether a the value at a given label is "concrete", i.e. whether it is completely independent of state and could be lifted wholesale out of the procedure body and would resolve to the same expression. Integer and string literals and protected global symbols fit this description.

6.3.3 Property dependencies

There is a more-or-less natural order in which we must analyze the properties of interest, for value-based properties, we usually must analyze **ExprseqLength**, for without being having solved results about expression sequences, we cannot hope to attempt to analyze function applications. Thus **SurfaceType** requires **ExprseqLength** to be executed first.

However, **SurfaceType** is also useful to **ExprseqLength** in certain contexts. Consider the following:

```
r := A[x];
```

Suppose ℓ is the annotation corresponding to A. In general, since A may be a table or array, little can be said about $\texttt{ES}(\ell)$. However, if it is known that $\texttt{STyp}(\ell) \sqsubseteq \{\texttt{STRING}, \texttt{LIST}\}$, and that $\texttt{STyp}(\ell) \sqsubseteq \{\texttt{INTPOS}\}$, then $\texttt{ES}(\ell) \sqsubseteq [1..1]$.

Thus we have a potential problem with *cyclic dependencies*. We resolve this in our software design by forcing dependencies to be DAG-like, but allow for a special "composite constraints" phase after all quantities have been analyzed in the current pass. This "composite constraints" phase is an extra traversal of the AST, in which any type of constraint may be freely used or generated.

Any constraints generated in this phase are included among the generated constraints in the next iteration.

6.3.4 Construction of property record

The following is a brief outline of the steps needed to turn a procedure p into a property record.

1. Check to see if p is a *built-in procedure* or special procedure; these have results in a special lookup table since they either cannot be analyzed, or are not analyzed for efficiency reasons.

2. Check the property database to determine if p has been examined before. If so, read any previously-analyzed property from the database.

3. Construct the annotated tree $\mathbf{AST}(p)$ and perform all preprocessing upon it.

4. Examine the dependency list for each of our analyzed properties, putting dependent analyses after their dependencies.

5. Analyze each property in turn, in the following way:

   (a) For each property p, generate its constraint set. If we have any "composite constraints" from a previous iteration, we add them now.

   (b) Write the constraint set to the property record.

   (c) Solution step:

   - Solve the constraint system if this is the first iteration or if the constraint system has changed since the last iteration.
   - If we already have a past solution for p, set it as our current solution. Otherwise, we set our solution to the most pessimistic possible ($\top$ for every quantity).
   - If we solve the system, we do so with the current solution as an initial condition.
   - Write the solution to the property record.

6. Generate "composite constraints": these are a special class of opportunistic constraints whose triggering condition depends on the results of several different analyses in a complex manner. We must therefore wait until all analyses have been completed before analyzing them.

7. If no fixed point has been generated, we repeat step 6.

8. If we have reached a fixed point, we return the current state of the property record as our solution.

Chapter 7

Results

After developing all this theoretical apparatus and design, we can now demonstrate some tangible examples of its and results on a nontrivial Maple programs.

We will begin with illustrations of the results of our analyses on small comprehensible Maple programs; some of these examples were also used in [3]. Following this will discuss two wide-scale deployments of our tool on libraries of Maple procedures.

7.1 Examples

The following three examples illustrate the use of our tool on small inputs which demonstrate the idea behind their use, the interaction between properties in solving a procedure, and the suitability of this tool for static error detection.

7.1.1 Example 1: Primality tester

It is helpful to begin with some concrete examples for which the analysis can be replicated by the reader. Consider the following Maple procedure:

```
IsPrime := proc(n::integer) local S, result;
   S := numtheory:-factorset(n);
   if nops(S) > 1 then
        result := (false, S);
   else
        result := true;
```

```
    end if;
    return(result);
end proc:
```

IsPrime is an combined primality tester and factorizer. It factors its input n, then returns a boolean result which indicates whether n is prime. If it is composite, the prime factors are also returned.

This small example demonstrates the results of two of our analyses. For **ExprseqLength** , we are able to conclude, even in the absence of any special knowledge or analysis of `numtheory:-factorset`, that `S` must be an expression because it is used in a call to the kernel function `nops` ("number of operands"); we glean this information from our function database.

Combined with the fact that `true` and `false` are known to be expressions, we can estimate the size of `result` as $[2..2]$ when the if-clause is satisfied and $[1..1]$ otherwise. Upon unifying the two branches, our **ExprseqLength** estimate for `result` becomes $[1..2]$.

Our results can also be used for static inference of programming errors. We assume that the code, as written, reflects the programmers' intent. In the presence of a programming error which is captured by one of our properties, the resulting constraint system will have trivial solutions or no solutions at all.

7.1.2 Example 2: GrowSeq

For a more complex example, consider:

```
GrowSeq := proc(u) local x, y, i;
     x := 2,3,4,5;
     y := 1;
     for i in [1,2,3] do
          x := x , u;
          y := y + 1;
     end do;
     (x, y);
end proc:
```

This example illustrates two key points: our capacity to deal with loop semantics, and our ability to propagate information between properties.

Initially, **ExprseqLength** measures the initial size of `x` as $[4..4]$, and attempts to solve the loop. However, it does not yet know how long the loop runs so it uses the extremely pessimistic estimate of $[0..\infty]$ steps. **LoopSteps** can measure recognize the size of the expression sequence inside the list [1,2,3]\ as [3..3], but cannot propagate this information anywhere.

Through the partition iteration approach, **LoopSteps** can see that the variable x is an expression sequence whose length is growing by 1 each step. Because of this initial pessimistic assessment of the number of loop steps, its first solution for the state of x at the end of the loop is $[3..\infty]$. **ExprseqLength** is not capable of doing anything with y because it only sees y as an expression.

Next, **SurfaceType** is able to solve the loop for y as it sees y as `INTPOS` and knows that Maple's sum operator produces an `INTPOS` when given two `INTPOS` expressions as input. Therefore we know now that y is an `INTPOS` throughout the loop.

Control passes to **NumberOfOperands**, which assigns the list [1,2,3]\ the measure $([1..1],[3..3])$.

Next, our "composite constraint" pass propagates the information about the size of the list into **LoopSteps**: we now know the loop takes exactly $[3..3]$ steps. This information is passes to **ExprseqLength** in the second pass, and because **SurfaceType** inferred that y was always an `INTPOS`, **ExprseqLength** now knows its size is $[1..1]$.

Eventually we exit with the estimate $[8..8]$ for the last expression in the procedure, having used five different analyses to obtain this result.

7.1.3 Example 3: Error detection

Our results can also be used for static inference of programming errors. We assume that the code, as written, reflects the programmers' intent. In the presence of a programming error which is captured by one of our properties, the resulting constraint system will have trivial solutions or no solutions at all.

For an illustration of this, consider the following example. The procedure `faulty` is bound to fail, as the arguments to `union` must be sets or unassigned names, not integers. As Maple is untyped, this problem will not be caught until runtime.

```
looptest := proc( n :: posint ) :: integer;
  local s :: integer, i :: integer, T :: table, flag :: true;
  ( s, i, flag ) := ( 0, 1, false );
  T := table();
  while i^2 < n do
     s := i + s;
     if flag then T[i] := s; end if;
     if type( s, 'even' ) then flag := true; break; end if;
     i := 1 + i
  end do;
  while type( i, 'posint' ) do
     if assigned(T[i]) then T[i] := T[i] - s; end if;
     if type( s, 'odd' ) then s := s - i^2 end if;
     i := i - 1
  end do;
  (s, T)
end proc:
```

Figure 7.1: Procedure `looptest` from test library

```
faulty := proc(c) local d, S;
    d := 1;
    S := {3,4,5};
    S union d;
end proc:
```

However, **SurfaceType** can detect this: the two earlier assignments impose the constraints $\texttt{STyp}(X_1) \sqsubseteq \{\texttt{INTPOS}\}$ and $\texttt{STyp}(X_2) \sqsubseteq \{\texttt{SET}\}$, while `union` imposes on its arguments the constraints that $X_3, X_4 \sqsubseteq \{\texttt{SET}\} \cup \mathrm{Alias}$(*Name*).

No assignments to `d` or `S` could have occurred in the interim, we also have the constraints $X_1 = X_4$ and $X_2 = X_3$. The resulting solution contains $X_1 = \emptyset$, which demonstrates that this code will always trigger an error.

7.2 Results from testing

We have completed two significant runs of our tools against collections of Maple procedures, the results of which we present below.

7.2.1 Example from compiler/partial evaluator test base

We have run our tools against a private collection of Maple functions gathered from earlier projects (including [4]); this should provide us with a solid test-bed which catches corner cases and tests the robustness of our design.

We can analyze 294 of the 301 procedures in this test base. The remaining seven cannot presently be analyzed by our tool because of technical details involving the manner in which lexically-scoped variables are retained inside Maple archives; the details of this issue are unrelated to our analysis.

Figure 7.2.1 is an example of a function present in the test library; we present a brief description of how our tool regards it.

This rather formidable procedure, while not doing anything particularly useful, is certainly complex. It contains two successive conditional loops which march in opposite directions, and both of which populate the table `T` along the way.

Here our analysis recognizes the fact that even though `flag` is written within the body of the first while loop, this write event cannot reach the if-condition on the preceding line because the write event is immediately followed by a `break` statement. We are also able to conclude that `s` is always an integer: though this is easy to see, given that all the write events to `s` are operations upon integer quantities.

7.2.2 Results from compiler/partial evaluator test base

We will now discuss the overall results from the test run. Figure 7.2 summarizes key information about the test run. First, for some point a in the abstract domain (e.g. $[1..1] \in \mathbb{I}(\mathbb{N})$, we ask the question for each test procedure p "how many times does a occur in our solution for p?".

We compute this, and then express it as a a ratio over all constraint variables used in p. This gives a sense of how frequently this measured value occurs in p. Finally, we compute the average of all such estimates over all procedures p in our test base, giving us a sense both of what is in a "typical" procedure and how effective we are at measuring it. Expressed formally,

The result is a rough guide to our precision. However, there is a significant degree of complexity in our chosen abstract domains. In the interests of simplifying some of this complexity and understanding how precise we are able to be, let us informally define a *precision measure* on the abstract domain which indicates how far we are from a concrete solution, independently of what the concrete solution actually is.

For the interval lattice $[a..b]$, we will define our precision measure to be

$$\mu(z) = \begin{cases} \infty & \text{if } z = [a..\infty] \text{ for some } a \\ b - a + 1 & \text{if } z = [a..b] \text{ for } a, b \in \mathbb{N} \\ 0 & \text{if } z = \bot \end{cases}$$

For the lattice of sets, our precision measure will simply be the cardinality of the set: $\mu(S) = |S|$. Observe in both cases that $\mu(\bot) = 0$ and that $\mu(x) = 1$ corresponds to an *atom*, which represents the most concrete solution possible. Quantities in the abstract domain whose μ values are equal have a comparable level of “concreteness,” and following this principle we have used the μ results to group related terms in Figure 7.2. In this figure we present the results from a traversal of the 294 procedures in this test base. We will proceed property by property:

ExprseqLength

Immediately we notice that on average we have $\mu(\texttt{ES}(\ell)) = 0$ approximately 0.43% of the time. These correspond to cases where $\texttt{ES}(\ell) = \bot$, which suggest either errors in the code or errors in our engine.

Either could be the case. If it is the result of an error in our inferencer, it is likely the result of a poorly-specified opportunistic rule, or a poorly-formulated entry in the function database.

We next observe that a huge proportion, an actual majority, of the values encountered turn out to be expressions. This represents what we can “prove” to be an expression; there may be other variables which turn out to be expressions but are currently more pessimistically classfied.

Together with expressions, expression sequences of lengths 0, 2, and 3 represent 70.0% of all values, and since completely-determined values make up 70.82%, we know there cannot be many bigger exprseqs in the code.

Next, notice that 28.2% of all values have infinite bounds. Of these 19.8% have $[0..\infty]$ as their bound, which is $\top$ in our lattice. These are values upon which we have made no progress at all. On the other hand, this means we have shown something nontrivial for 80.2% of values.

ExprseqLength	% match
$\mu(\texttt{ES}(\ell)) = 0$	0.43
$\mu(\texttt{ES}(\ell)) = 1$	70.82
$\texttt{ES}(\ell) = [0..0]$	1.190
$\texttt{ES}(\ell) = [1..1]$	66.1
$\texttt{ES}(\ell) = [2..2]$	2.68
$\texttt{ES}(\ell) = [3..3]$	0.563
$\mu(\texttt{ES}(\ell)) = 2$	0.508
$\texttt{ES}(\ell) = [0..1]$	0.122
$\texttt{ES}(\ell) = [1..2]$	0.374
$\texttt{ES}(\ell) = [2..3]$	0.016
$\mu(\texttt{ES}(\ell)) = 3$	0.0146
$\mu(\texttt{ES}(\ell)) = 4$	0
$\mu(\texttt{ES}(\ell)) = \infty$	28.2
$\texttt{ES}(\ell) = [0..\infty]$	19.76
$\texttt{ES}(\ell) = [1..\infty]$	7.06
$\texttt{ES}(\ell) = [2..\infty]$	0.92
$\texttt{ES}(\ell) = [3..\infty]$	0.39

SurfaceType	% match
$\mu(\texttt{STyp}(\ell)) = 0$	7.27
$\mu(\texttt{STyp}(\ell)) = 1$	25.0
$\texttt{STyp}(\ell) = \{\texttt{INTPOS}\}$	12.77
$\texttt{STyp}(\ell) = \{\texttt{STRING}\}$	0.347
$\texttt{STyp}(\ell) = \{\texttt{INTNEG}\}$	1.56
$\texttt{STyp}(\ell) = \{\texttt{NAME}\}$	1.51
$\mu(\texttt{STyp}(\ell)) = 2$	0.72
$\texttt{STyp}(\ell) = \text{Alias}($*Integer*$)$	0.289
$\mu(\texttt{STyp}(\ell)) = 3$	0.248
$\mu(\texttt{STyp}(\ell)) = 4$	0.0400
$\mu(\texttt{STyp}(\ell)) = 5$	0.00
$\mu(\texttt{STyp}(\ell)) = 6$	3.43
$\texttt{STyp}(\ell) = \text{Alias}($*Algebraic*$)$	16.50
$\texttt{STyp}(\ell) = \text{Alias}($*Complex*$)$	21.45
$\texttt{STyp}(\ell) = \text{Alias}($*Expression*$)$	21.45
$\texttt{STyp}(\ell) = \text{Alias}($*AnyValue*$)$	21.74

NumOperands	% match
$\texttt{NOps}(\ell) = ([a..b], ?)$	34.80
$\texttt{NOps}(\ell) = [0..0]$	0
$\texttt{NOps}(\ell) = [1..1]$	27.8
$\texttt{NOps}(\ell) = [2..2]$	2.77
$\texttt{NOps}(\ell) = [3..3]$	0.05
$\mu(\texttt{NOps}(\ell)) = 2$	0
$\mu(\texttt{NOps}(\ell)) = 3$	0
$\mu(\texttt{NOps}(\ell)) = 4$	0
$\mu(\texttt{NOps}(\ell)) = \infty$	34.5
$\texttt{NOps}(\ell) = [0..\infty]$	32.08
$\texttt{NOps}(\ell) = [1..\infty]$	0.89
$\texttt{NOps}(\ell) = [2..\infty]$	0.91
$\texttt{NOps}(\ell) = [3..\infty]$	0.35

LiteralValue	% match
$\texttt{LVal}(\ell) = (x, ?)$	77.4
$\mu(\texttt{LVal}(\ell)) = 0$	0.00
$\mu(\texttt{LVal}(\ell)) = 1$	22.2
$\texttt{LVal}(\ell) = \{\texttt{true}\}$	2.02
$\texttt{LVal}(\ell) = \{\texttt{false}\}$	0.05
$\texttt{LVal}(\ell) = \{-1\}$	1.667
$\texttt{LVal}(\ell) = \{0\}$	6.23
$\texttt{LVal}(\ell) = \{\frac{1}{2}\}$	0.01
$\texttt{LVal}(\ell) = \{1\}$	3.96
$\texttt{LVal}(\ell) = \{2\}$	0.553
$\texttt{LVal}(\ell) = \{3\}$	2.02
$\texttt{LVal}(\ell) = \{4\}$	2.02
$\mu(\texttt{LVal}(\ell)) = 2$	0.34
$\texttt{LVal}(\ell) = \{\texttt{true}, \texttt{false}\}$	0.31
$\mu(\texttt{LVal}(\ell)) = 3$	0.04

Figure 7.2: Results using compiler/partial evaluator test base

SurfaceType

Our error-set is much larger here: 7.27%. This is perhaps the result of errors in the code or a large quantity of dead code; another explanation is an inferencer bug somewhere.

We see that we can assign a unique surface type to 25% of values, and that positive integers represent over half of this total: integers are truly ubiquitous in Maple.

Lastly, quite a number of matches come up for our type aliases *Algebraic*, *Complex*, etc. As we use these in the function database, it is likely this information is merely being propagated from there.

NumOperands

Here we see first that 34% of values are not provably expressions as far as the **NumOperands** analysis is concerned; this accords well with the conclusion from **ExprseqLength** that 66% of values were expressions.

Of the 66% that are expressions, approximately 32% have finite bounds, while the remaining do not. Our rate of returning $\top$ is higher than it was for $\mathbf{ExprseqLength}$, but this could also reflect the more complex semantics of $\mathbf{NumberOfOperands}$.

LiteralValue

Lastly, we examine literal values. Unsurprisingly, 77.4% of values cannot be assigned a literal value; this is hardly shocking news since we are dealing with a *very* concrete property.

Almost all the literal value sets that can we have encountered are singletons. This probably reflects an opportunistic constraint assigning a value to a literal in context, and that information propagating from there.

We note that the literal values encountered are the integers -1,0,1 and “true” are particularly well-represented in the results; this is not especially surprising, especially given the widespread use in the Maple library of the `sign` and `signum` commands.

7.2.3 Results from Maple library test base

The obvious candidate for a Maple library to use as a data mine for testing purposes is the Maple library itself. Figure 7.3 presents the results from a traversal of 116 procedures chosen semi-randomly from the Maple 10 standard library. (We say “semi-randomly” because we required that the procedures chosen have certain upper bounds on size, complexity, and level of dependence on other procedures in order that we might analyze them).

ExprseqLength	% match
$\mu(\texttt{ES}(\ell)) = 0$	0.849
$\mu(\texttt{ES}(\ell)) = 1$	63.78
$\texttt{ES}(\ell) = [0..0]$	3.75
$\texttt{ES}(\ell) = [1..1]$	57.5
$\texttt{ES}(\ell) = [2..2]$	0.255
$\texttt{ES}(\ell) = [3..3]$	1.64
$\mu(\texttt{ES}(\ell)) = 2$	0.511
$\texttt{ES}(\ell) = [0..1]$	0.090
$\texttt{ES}(\ell) = [1..2]$	0.399
$\texttt{ES}(\ell) = [2..3]$	0.000
$\mu(\texttt{ES}(\ell)) = 3$	0
$\mu(\texttt{ES}(\ell)) = 4$	0
$\mu(\texttt{ES}(\ell)) = \infty$	35.36
$\texttt{ES}(\ell) = [0..\infty]$	28.09
$\texttt{ES}(\ell) = [1..\infty]$	4.71
$\texttt{ES}(\ell) = [2..\infty]$	1.7
$\texttt{ES}(\ell) = [3..\infty]$	0.45

SurfaceType	% match
$\mu(\texttt{STyp}(\ell)) = 0$	1.07
$\mu(\texttt{STyp}(\ell)) = 1$	26.07
$\texttt{STyp}(\ell) = \{\texttt{INTPOS}\}$	10.9
$\texttt{STyp}(\ell) = \{\texttt{STRING}\}$	4.01
$\texttt{STyp}(\ell) = \{\texttt{INTNEG}\}$	0.86
$\texttt{STyp}(\ell) = \{\texttt{NAME}\}$	0.01
$\mu(\texttt{STyp}(\ell)) = 2$	0.12
$\texttt{STyp}(\ell) = \text{Alias}(\textit{Integer})$	0.289
$\mu(\texttt{STyp}(\ell)) = 3$	0.00
$\mu(\texttt{STyp}(\ell)) = 6$	1.218
$\texttt{STyp}(\ell) = \text{Alias}(\textit{Complex})$	1.2
$\texttt{STyp}(\ell) = \text{Alias}(\textit{Expression})$	36.16
$\texttt{STyp}(\ell) = \text{Alias}(\textit{AnyValue})$	30.0

NumOperands	% match
$\texttt{NOps}(\ell) = ([a..b], ?)$	43.56
$\mu(\texttt{NOps}(\ell)) = 1$	30.15
$\texttt{NOps}(\ell) = [0..0]$	0.503
$\texttt{NOps}(\ell) = [1..1]$	28.4
$\texttt{NOps}(\ell) = [2..2]$	1.19
$\texttt{NOps}(\ell) = [3..3]$	0.00
$\mu(\texttt{NOps}(\ell)) = 2$	0.12
$\texttt{NOps}(\ell) = [1..2]$	0.12
$\mu(\texttt{NOps}(\ell)) = \infty$	26.18
$\texttt{NOps}(\ell) = [0..\infty]$	26.18

LiteralValue	% match
$\texttt{LVal}(\ell) = (x, ?)$	79.29
$\mu(\texttt{LVal}(\ell)) = 0$	0.00
$\mu(\texttt{LVal}(\ell)) = 1$	20.7
$\texttt{LVal}(\ell) = \{\texttt{true}\}$	0.007
$\texttt{LVal}(\ell) = \{\texttt{false}\}$	0.05
$\texttt{LVal}(\ell) = \{-1\}$	0.983
$\texttt{LVal}(\ell) = \{0\}$	6.23
$\texttt{LVal}(\ell) = \{\frac{1}{2}\}$	0.099
$\texttt{LVal}(\ell) = \{1\}$	0.983
$\mu(\texttt{LVal}(\ell)) = 2$	0
$\mu(\texttt{LVal}(\ell)) = 3$	0.0431

Figure 7.3: Results using Maple library as a test base

The presentation is identical to that for Figure 7.2. We shall therefore not repeat ourselves excessively and concentrate on the differences.

ExprseqLength

The results are in the same general bounds, but it is worth observing that the rate of encountering $\perp$ has doubled. It will require some investigation to determine whether this is a genuine result or an inferencer error.

Generally, the results are somewhat poorer: there are more quantities with $\mu(\mathtt{ES}(\ell)) = \infty$, and fewer proven expressions.

SurfaceType

The results are slightly weaker here as well, though the exceptionally high rate of returning $\perp$ exhibited in the previous analysis is fortunately not replicated.

NumOperands

Fully 44% of values cannot be proven to be expressions, and of those that can be shown to be expressions, many of them cannot be given a finite bound.

LiteralValue

The results here are essentially equivalent to those from the previous analysis, with integers, other numeric literals, and symbols making up the bulk of "literal quantities" we are able to discover.

7.2.4 Discussion

The results above suggest future directions for this analyzer. A special focus on **SurfaceType** would probably be fruitful, as it is clear that many of the constraints visible there are not being propagated forwards through chains of assignments and disseminated over other constraint variables.

This suggests we have a phenomenon whereby there is a island of pure concrete knowledge lost in a sea of approximation and uncertainty: the answer is to focus not on the depth of the quantities we wish to impose, but on their breadth. We should ensure there is nothing about which the system is largely ignorant.

That said, there is a large body of code we will likely never be able to analyze effectively: this includes such things as dynamic variable generators, and dynamically-generated procedures. One clearly beneficial addition would be a means of authoritatively knowing when a symbolic quantity was an assigned value and when it is not.

Chapter 8

Conclusion

We have demonstrated that static analysis and specifically abstract interpretation can be a highly informative tool for inferring static information about Maple code. Each of the chief value-based properties of interest have nontrivial amounts of inferred data for even small input procedures.

Our tool is suitably generic, and can handle a wide class of Maple inputs while generating nontrivial results. Our efforts at avoidance of the "toy problem" syndrome have, on balance, succeeded: while the solver is extremely slow for large procedures, this is to probably be expected for this type of analysis, and the extreme genericity of our implementation means many efficiency improvements are possible.

The approach of employing a small number of specialized analyses which compose well has proven to be a very successful strategy, as the example from 7.1.2 illustrates. We have seen that the techniques of static inference through abstract interpretation permit us to reclaim some of the knowledge which in other programming languages we get for free.

This tool could be combined with other tools with good results, like compilers, code optimizers, or partial evaluators. As discussed in the introduction, such tools could make use of the inferred static data in much the same way as they might make use of inferred types. Some errors can be caught; some dead code can be eliminated; some specializations or code transformations are enabled by the existence of such information.

There are many possible directions for future work in this area. An obvious candidate is efficiency improvements to the constraint solver. The solver is currently extremely generic; while we do not want to specialize our constraint solution techniques, we can undoubtedly make the solution engine much, much faster and scaleable.

Another clear target for further improvement is the addition of more static properties; like the ones described here, these would have to be simple yet compose well with other existing properties. We might, for instance, have a “integer arithmetic” solver which models integer-valued variables with intervals. This would scale better than our **LiteralValue** analysis, and has the potential for sharing data usefully with the other interval-valued properties, like **ExprseqLength** and **NumberOfOperands**.

Our ability to solve recurrence relations over lattices, even in restricted cases, offers up intriguing possibilities regarding the potential applicability of symbolic tools in abstract interpretation. We suspect that the class of solveable recurrences over our abstract domains can be extended considerably with some further work, and it is possible that this may permit us to recapture some precision that would otherwise be lost with a straightforward widening/narrowing approach.

Finally, the rulesets for opportunistic constraint assignment could be vastly intended, permitting the refinement of existing analyses. The opportunistic rules used thus far are only a small selection of the data that could be drawn from the code. Furthermore, the current implementation significantly underutilizes the potential of the so-called “composite constraints”, that is, opportunistic constraint assignment rules which depend on one or more existing analyses to be already present.

Appendix A

Inert Form Tags

This is a list of all tag names which occur in Maple's inert form data structure, with a brief explanation of each.

We have classified the tags into three groups. The first two groups correspond to *statements* and *expressions* respectively. The third corresponds neither to procedures nor expressions, but merely data within the AST.

A.1 Inert Forms Corresponding to Statements

Table A.1: Inert Form Tags Corresponding to Statements

Inert Form	Description	Children in AST
ASSIGN	Assignment operator	2
BREAK	Break out of loop	0
CONDPAIR	If condition and associated branch	2
ERROR	Raise error	1
FORFROM	For-from loop	6
FORIN	For-in loop	4
IF	If statement	Arbitrary
NEXT	Jump to next loop iteration	0
STATSEQ	Sequence of statements	Arbitrary
STOP	Terminate session	0
TRY	Try/catch/finally block	Arbitrary
RETURN	Quit procedure and return value	1
READ	Read data from external archive	1
SAVE	Save data to external archive	1

A.2 Inert Forms Corresponding to Values

Table A.2: Inert Forms Corresponding to Values

Inert Form	Description	Children in AST
AND	Boolean conjunction ($\wedge$)	2
ASSIGNEDNAME	Assigned name	2 or 3
ARRAY	Rectangular array of data	5 or 6
ARGS	Expseq of dynamic arguments	0
CACHETAB	Special memoization table	3
CATENATE	Concatenate two strings or names	2
COMPLEX	Complex number	1 or 2
DCOLON	Check type of expression	2
EQUATION	equation ($=$)	2
EXPSEQ	expression sequence	Arbitrary
EXACTSERIES	Mathematical series w/no order term	Arbitrary
FLOAT	Floating-point number	2
FUNCTION	Function application	2
HASHTAB	Hash table	Arbitrary
INTPOS	Nonnegative integer	0
INTNEG	Negative integer	0
INEQUAT	Inequation ($\neq$)	2
IMPLIES	Boolean implication ($\Rightarrow$)	2
LESSEQ	Less than or equal to ($\leq$)	2
LESSTHAN	Less than ($<$)	2
LIST	List of expressions	1
LOCAL	Local variable in module or procedure	1
LOCALNAME	Local value	2
LEXICAL_LOCAL	Local variable from higher lexical scope	1
LEXICAL_PARAM	Procedure parameter from higher lexical scope	1
MATRIX	Matrix data structure	5 or 6
MEMBER	Module member reference	2
MODDEF	Software module definition	9
MODULE	Software module	3
NAME	Variable name	1 or 2

NARGS	Number of dynamic procedure arguments	0
NOT	Boolean negation ($\neg$)	1
NRESULTS	Number of results expected from application of procedure	0
OR	Boolean disjunction ($\vee$)	2
PARAM	Procedure parameter	1
POWER	Exponent data structure	2
PROC	Procedure	9 or 10
PROD	Product data sructure	Arbitrary
PROCNAME	Special name for procedure self-reference	0
RANGE	A range $[a..b]$	2
RATIONAL	Fractional number	2
SDPOLY	Special data structure for sparse distributed multivariate polynomial	7
SERIES	A mathematical series approximation	Arbitrary
SET	Set of expressions	1
STRING	String	0
SUM	Sum data structure	Arbitrary
TABLE	Hash table	2
TABLEREF	Indexed expression	2
UNEVAL	Evaluation delay operator	1
VECTOR_COLUMN	Column vector	5 or 6
VECTOR	Vector of unspecified orientation	5 or 6
VECTOR_ROW	Row vector	5 or 6
XOR	Boolean exclusive or ($\oplus$)	2
ZPPOLY	Special polynomial structure for computations modulo p	7

So as not to make the table excessively verbose, we no not list those tags which relate to the special parameter processing rules introduced in Maple 10. Because these features are so new, they are effectively never encountered in analysis of existing code.

Nevertheless, for completeness' sake, the names of the omitted tags are:

NOPTIONS	NPARAMS	NREST	OPTIONS	PARAMS	REST

A.3 Additional Inert Form Tags

There are a small number of remaining inert form tags. These tags can neither be regarded as statements or values, but serve as supplemental data to some other type of inert form. One demonstration of this is that fact that these additional tags occur within extremely specific contexts, such as an inert procedure, module, or hash table. They have no parallel by themselves in the world of “live expressions”.

Table A.3: Unclassified Inert Form Tags

Inert Form	Description	Context Appearing
ATTRIBUTE	Container for Maple attributes	NAME, ASSIGNEDNAME
DESCRIPTIONSEQ	Container for descriptive text	MODULE, PROC
EOP	Parameter order evaluation data	PROC
EXPORTSEQ	Names exported from a module	MODULE
GLOBALSEQ	Global names used in module or procedure body	MODULE, PROCS
HASHPAIR	Key/value pairs in hashtable	HASHTAB
LOCALSEQ	Local variable names	MODULE, PROC
LEXICALPAIR	A lexical binding	MODULE, PROC
LEXICALSEQ	List of lexical bindings	MODULE, PROC
OPTIONSEQ	List of options specified	MODULE, PROC
PARAMSEQ	Procedure parameter names	MODULE, PROC
RETURNTYPE	Return type for procedure	PROC

Appendix B

Surface Type Aliases

In the design of the system for opportunistic constraint generation (see Section 6.1), which relies on analysis of inert forms, it frequently became necessary to refer repeatedly to particular *sets* of inert tags which had similar semantics or characteristics.

As these particular sets of tags were often large, it became convenient and ultimately necessary to invent a system of *aliases* to abbreviate them. The names chosen reflect the meaning attached to this class of inert forms in Maple; in many cases the name is identical in name and meaning to identical to one of Maple's built-in types (see Section 2.3).

Table B.1 catalogues the important aliases used in Section 6. They should not be regarded as "results" per se, but the fact that they were created out of necessity to capture the semantics shows they have some significance.

Figure B.1 illustrates the hierarchical relationship between several of the aliases.

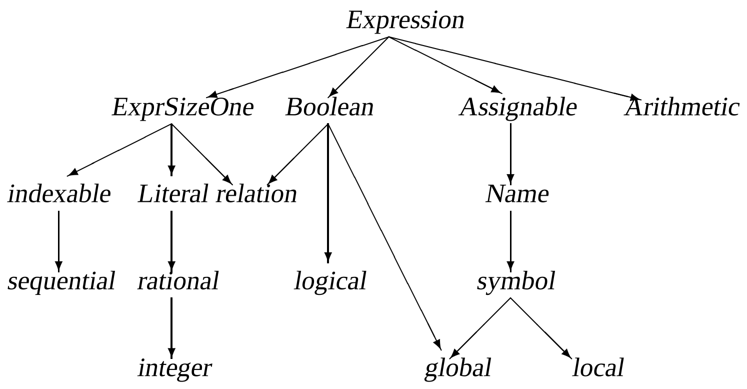

Figure B.1: Relationships between type aliases

Table B.1: Type Alias Names

Alias Name	**Definition**
integer	{INTPOS, INTNEG}
rational	{RATIONAL} ∪ *integer*
float	{FLOAT, HFLOAT}
Numeric	*float* ∪ *rational*
Literal	{COMPLEX, STRING} ∪ *Numeric*
relation	{EQUATION, INEQUAT, LESSEQ, LESSTHAN}
logical	{IMPLIES, AND, NOT, OR, XOR}
Bool	{NAME} ∪ *relation* ∪ *logical*
global	{NAME, ASSIGNEDNAME}
local	{LOCALNAME, ASSIGNEDLOCALNAME}
symbol	*local* ∪ *global*
Name	{TABLEREF} ∪ *symbol*
Assignable	{FUNCTION} ∪ *Name*
Arithmetic	{PROD, SUM, POWER}
sequential	{SET, LIST}
last_name_eval	{PROC, MODULE, TABLE}
WeirdAlgebraic	{ZPPOLY, SDPOLY, EXACTSERIES, SERIES}
Algebraic	*WeirdAlgebraic* ∪ *Assignable* ∪ *Arithmetic*
Vector	{VECTOR, VECTOR_COLUMN, VECTOR_ROW}
tabular	{ARRAY, MATRIX, TABLE} ∪ *Vector*
indexable	*tabular* ∪ *sequential* ∪ {STRING}
ExprSizeOne	{DCOLON, RANGE, MODDEF} ∪ *WeirdAlgebraic* ∪ *indexable* ∪*Literal* ∪ *relation* ∪ *last_name_eval*
Expression	{UNEVAL} ∪ *Bool* ∪ *ExprSizeOne* ∪ *Assignable* ∪ *Arithmetic*
EvalAwaySymbol	{PROCNAME, MEMBER, PROCNAME, MEMBER, LOCAL, LEXICAL_LOCAL, PARAM, LEXICAL_PARAM}
ProcOps	{ARGS, RESULTS, OPTIONS, PARAMS, REST}
NumProcOps	{NARGS, NRESULTS, NOPTIONS, NPARAMS, NREST}
EvalAway	*EvalAwaySymbol* ∪ *ProcOps* ∪ *NumProcOps*
AnyValue	*Expression* ∪ *EvalAway*
Statement	{ASSIGN, BREAK, CONDPAIR, ERROR, FORFROM, FORIN, IF, NEXT, RETURN, READ, SAVE, STATSEQ, STOP, TRY}

Appendix C

Opportunistic constraint rules

Here we list a few of the "opportunistic rules" for constraint generation while traversing the AST. As the system has hundreds of such rules, this listing should not be regarded as anywhere near complete.

Example	Rule Trigger	Constraint	Description
x,y,z := e;	Simultaneous assignment to n variables with $n \geq 2$	$\mathtt{ES}(\mathtt{e}) = [n..n]$	Right-hand side must have size n or assignment will fail.
PARAM(7)	Occurrence of nth procedure parameter	$\mathtt{ES}(\mathtt{ARGS}) \sqsubseteq [n..\infty]$	# of arguments must be $\geq n$.
r := [e];	List construction	$\mathtt{NOps}(\mathtt{r}) = ([1..1], \mathtt{ES}(\mathtt{e}))$	Size of list is size of underlying expseq.
for x in L do	Loop iteration over expression	$\mathtt{LSteps}(\ell) \sqsubseteq \mathtt{NOps}(L)$	# of loop steps bounded by # of operands of `L`.
if c then	If condition	$\mathtt{STyp}(\mathtt{c}) \sqsubseteq \mathrm{Alias}(\mathit{Bool})$	Conditional expression must have "boolean" surface type.
x := a;	Assignment	$\#\mathtt{W}(\mathtt{x}) = [1..1]$	x is written to

Appendix D

Special Function Table

Following is included the source code for our database of built-in functions. Though some the content is infrastructure for fetching and retrieving data, a typical entry looks something like the following:

```
T["assigned"]:= {"ES" = {"r" = 1..1, 0 = 1..1},
                 "ST" = {"r" = {MIF(NAME)}},
                 "LV" = { "r" = {_Inert_NAME("true"),
                                 _Inert_NAME("false")}},
                 "Pure" = false, "Builtin" = true }:
```

The abbreviations ES, ST, PV indicate specific static properties (namely **ExprseqLength**, **SurfaceType**, and **LiteralValue**). These abbreviations appear in the entry in the form *prop_name*=S, where S is a set which contains information relevant to property *prop_name*.

In the set S are additional equations, whose left-hand side consists of the string "r" or nonnegative integers. These refer to quantities in a function call:

- "r" refers to the return value
- 0 returns to the argument expseq as a whole
- A positive integer n refers to the nth argument

The right-hand side of this inner quantity refers to the value by which the function argument or return value should be constrained.

Therefore, the first line {"ES" = {"r" = 1..1, 0 = 1..1} of our example states that when encountering the function "assigned", its return value and argument sequence should each be $\sqsubseteq [1..1]$ for expseq size.

N.B. the name `MIF(foo)` is an abbreviation for `_Inert_foo`, while `TA(bar)` refers to the "type alias" `bar` (see Appendix B).

```
FunctionData:= module() export Get, Set, Select, Defined, Data;
    local
        T, classtab, defaultval,
        GenST, EStoST,
        u, v;

    Data := proc(fname::string) local es, st, na, lv, i,
                                        L_st, L_na, L_lv;
        es := table( Get( fname, "ES" ) );

        st := table( Get( fname, "ST" ) );
        L_st := select( type, DataMap:-Keys(st), 'posint' );

        na := Get( fname, "NA" );
        na := table( `if`(na=FAIL,[],na) );
        L_na := select( type, DataMap:-Keys(na), 'posint' );

        lv := Get( fname, "LV" );
        lv := table( `if`(lv=FAIL,[],lv) );
        L_lv := select( type, DataMap:-Keys(lv), 'posint' );

        table([
            "AT" = Annotate( eval( convert(fname, 'symbol') ) ),

            "S_ES" = table([
                ProcFV(0)     = 'CL_ExprSize'(es["r"]),
                ProcIV(_Inert_ARGS(0),0) = 'CL_ExprSize'(es[0])
            ]),

            "S_NA" = table([
                ProcFV(0)='CL_ExprNopsSize'(
                    es["r"],
                    `if`(assigned(na["r"]),na["r"],NULL)
                ),
                ProcIV('_Inert_ARGS'(0),0) =
                            CL_ExprNopsSize(es[0]),
                seq(
                    ProcIV('_Inert_PARAM'(i,0),0)
                                = CL_ExprNopsSize(es[i]),
                    i = L_na
                )
            ]),

            "S_ST" = table([
                ProcFV(0) = 'CL_SurfaceType'(st["r"]),
                ProcIV('_Inert_ARGS'(0),0)='CL_SurfaceType'(st[0]),
                seq(
                    ProcIV('_Inert_PARAM'(i,0),0)
                                ='CL_SurfaceType'(st[i]),
                    i = L_st
                )
            ]),

            "S_LV" = table([
                ProcFV(0) = 'CL_LiteralValue'(
                    st["r"],
                    `if`(assigned(lv["r"]),lv["r"],NULL)
                ),
                ProcIV(_Inert_ARGS(0),0) =
                        'CL_LiteralValue'(st[0]),
                seq(
                    ProcIV(_Inert_PARAM(i,0),0) =
                            'CL_LiteralValue'(st[i]),
                    i = L_lv
                )
            ])
        ])
    end proc:

    # Tell whether a given name exists in our 'database'
    Defined := proc(fname::string) assigned(T[fname]) end proc:

    Get := proc(fname::string, quan::string) local v, cls, fdata;
        if not assigned(T[fname]) then return(FAIL); end if;
        fdata := table( T[fname] );

        v := ADataMap:-Lookup( quan, T[fname] );
        if v <> FAIL then return(v); end if;

        cls := ADataMap:-Lookup( "class", T[fname] );
        if cls <> FAIL then
            v := ADataMap:-Lookup( quan, classtab[cls] );
            if v <> FAIL then return(v); end if;
        else
            ADataMap:-Lookup( quan, defaultval );
        end if;
    end proc:

    Set := proc(fname::string, quan::string, data)
        T[fname] := ADataMap:-Insert( quan, data, T[fname] );
    end proc:

    Select := proc(quan::string, val) local p;
        p := (fname)->`if`(val=Get(fname,quan),fname,NULL):
        map( p, DataMap:-Keys(T) );
    end proc:

    EStoST := proc(es::Or(range,identical(FAIL)), st1) local st;
        st := `if`( nargs > 1, st1, TA("Expression") );
        if es=0..0 or lhs(es) >= 2 then {MIF(EXPSEQ)}
        elif es=1..1 then st
        else {MIF(EXPSEQ)} union st; end if;
    end proc:

    GenST := proc(fname::string)
        local es, esr, es0, st, str, st0;

        es := Get( fname, "ES" );
        esr := ADataMap:-Lookup( "r", es );
        if esr='FAIL' then esr := 0..infinity; end if;
        if esr=0..0 then
            Set(fname, "LV", {"r" = {_Inert_EXPSEQ()}});
        end if;
        es0 := ADataMap:-Lookup(  0,  es );
        if es0='FAIL' then es0 := 0..infinity; end if;

        st := Get( fname, "ST" );
        str := ADataMap:-Lookup( "r", st );
        str := `if`( str <> FAIL, EStoST(esr, str), EStoST(esr) );
        st0 := ADataMap:-Lookup( 1, st );
        st0 := `if`( st0 <> FAIL, EStoST(es0, st0), EStoST(es0) );

        st := ADataMap:-Insert( 0,  st0, st);
        st := ADataMap:-Insert("r", str, st);

        Set(fname, "ST", st);
    end proc:

    classtab := table([]);
    T        := table([]);

    classtab["Hyperbolic Trig"] :={"ES" = {"r" = 1..1, 0 = 1..1},
                                   "ST" = {"r" = TA("Algebraic"),
                                            1  = TA("Algebraic")},
                                   "Pure" = true, "Builtin" =false}:
    classtab["Trigonometric"]  := {"ES" = {"r" = 1..1, 0 = 1..1},
                                  "ST" = {"r" = TA("Algebraic"),
                                           1  = TA("Algebraic")},
                                   "Pure" = true, "Builtin" =false}:
    classtab["Elementary"] := {"ES" = {"r" = 1..1, 0 = 1..1},
                               "ST" = {"r" = TA("Algebraic"),
                                        1  = TA("Algebraic")},
                              "Pure" = true, "Builtin" =false}:
    classtab["relation"]   := {"ES" = {"r" = 1..1, 0 = 2..2},
                              "Pure" = true, "Builtin" =true}:
    classtab["logical"]    := {"Pure" = true, "Builtin" =true}:

    # this is the default when something is entirely unspecified.
    defaultval := { "Pure"    = false,
                    "IO"      = false,
                    "Builtin" = false,
                    "ES"      = {"r" = 0..infinity, 0 = 0..infinity},
                    "ST"      = {"r" = TA("AnyValue") } }:

    T["$"]     := { "ES" = {"r"=0..infinity, 0=1..2},
                    "ST" = {"r"=TA("AnyValue"),
                            1 = {MIF(RANGE)},
                            2 = TA("Assignable") union
                                TA("integer") union
```

```
                                   {MIF(EQUATION)}},
                       "Pure" = false, "Builtin" = true }:

T["^"]        := { "ES" = {"r"=1..infinity, 0 = 2..2},
                   "Pure" = true, "Builtin" = true }:
T["*"]        := { "ES" = {"r" = 1..1, 0 = 0..infinity},
                   "Pure" = true, "Builtin" = true }:
T["**"]       := { "ES" = {"r" = 1..infinity, 0 = 2..2},
                   "Pure" = true, "Builtin" = true }:
T["+"]        := { "ES" = {"r" = 1..1, 0 = 0..infinity},
                   "Pure" = true, "Builtin" = true }:

T[".."]       := { "ES" = {"r"=1..1, 0=2..2},
                   "ST" = {"r"={MIF(RANGE)}},
                   "Pure" = true, "Builtin" = true }:

T["<"]        := { "ST" = {"r"={MIF(LESSTHAN)}},
                   "class" = "relation" }:
T["<="]       := { "ST" = {"r"={MIF(LESSEQ)}},
                   "class" = "relation" }:
T["<>"]       := { "ST" = {"r"={MIF(INEQUAT)}},
                   "class" = "relation" }:
T["="]        := { "ST" = {"r"={MIF(EQUATION)}},
                   "class" = "relation" }:
T[">"]        := { "ST" = {"r"={MIF(LESSTHAN)}},
                   "class" = "relation" }:
T[">="]       := { "ST" = {"r"={MIF(LESSEQ)}},
                   "class" = "relation" }:

T["?()"]      := { "ES" = {"r"=0..infinity, 0=1..infinity},
                              "Pure" = true, "Builtin" = true }:
T["?[]"]      := { "ES" = {"r"=0..infinity, 0=2..3},
                   "Pure" = true, "Builtin" = true }:

T["abs"]      := { "ES" = {"r" = 1..1, 0=1..2},
                   "Pure" = true, "Builtin" = true }:

T["add"]      := { "ES" = {"r" = 0..infinity, 0 = 2..2},
                   "ST" = {"r" = TA("Algebraic"),
                           2  = {MIF(EQUATION),
                                    MIF(FUNCTION)}}  }:

T["and"]     := { "ES" = {"r" = 1..1, 0 = 2..2},
                  "ST" = {"r" = TA("Assignable")
                                union TA("Boolean"),
                          1   = TA("Assignable")
                                union TA("Boolean"),
                          2   = TA("Assignable")
                                union TA("Boolean")},
                  "class" = "logical" }:
T["arcsin"]    := { "class" = "Trigonometric" }:
T["arccos"]    := { "class" = "Trigonometric" }:
T["arctan"]    := { "class" = "Trigonometric" }:
T["arcsec"]    := { "class" = "Trigonometric" }:
T["arccsc"]    := { "class" = "Trigonometric" }:
T["arccot"]    := { "class" = "Trigonometric" }:
T["arcsinh"]    := { "class" = "Trigonometric" }:
T["arccosh"]    := { "class" = "Trigonometric" }:
T["arctanh"]    := { "class" = "Trigonometric" }:
T["arcsech"]    := { "class" = "Trigonometric" }:
T["arccsch"]    := { "class" = "Trigonometric" }:
T["arccoth"]    := { "class" = "Trigonometric" }:
T["arctan"]     := { "ES" = {"r" = 1..1, 0 = 1..2},
                     "ST" = {"r" = TA("Algebraic"),
                             1   = TA("Algebraic"),
                             1   = TA("Algebraic")},
                        "Pure" = true, "Builtin" = false }:

# should handle in RD
T["array"] := { "ES" = {"r" = 1..1, 0 = 1..infinity},
                "ST" = {"r" = {MIF(TABLE)}}, # truefalse
                "Pure" = true, "Builtin" = true }:
T["Array"] := { "ES" = {"r" = 1..1, 0 = 1..infinity},
                "ST" = {"r" = {MIF(ARRAY)}}, # truefalse
                "Pure" = true, "Builtin" = true }:
T["assign"] := { "ES" = {"r" = 0..0, 0 = 0..infinity},
                "Pure" = false, "Builtin" = true }:

T["assigned"]:= {"ES" = {"r" = 1..1, 0 = 1..1},
                "ST" = {"r" = {MIF(NAME)}},
                 "LV" = { "r" = {_Inert_NAME("true"),
                                 _Inert_NAME("false")}},
                 "Pure" = false, "Builtin" = true }:

T["ASSERT"] := { "ES" = {"r" = 0..0, 0 = 0..infinity},
                 "Pure" = true, "Builtin" = true }:

T["cat"]    := { "ES" = {"r" = 1..1, 0 = 0..infinity},
                "ST" = {"r" = TA("name/string")
                              union {MIF(CATENATE)}}}:

# ST_imp (acts like identity on ints)
T["ceil"]   := { "ES" = {"r" = 1..1, 0 = 1..2},
                "ST" = {"r" = {MIF(FUNCTION)} union TA("integer")},
                "Pure" = true, "Builtin" = true }:
T["coeff"]  := { "ES" = {"r" = 1..1, 0 = 2..3},
                "Pure" = true, "Builtin" = true }:
T["convert"] := { "ES" = {"r" = 1..1, 0 = 2..infinity},
                "Pure" = true, "Builtin" = true }:

T["cos"]    := { "class" = "Trigonometric" }:
T["cosh"]   := { "class" = "Hyperbolic Trig" }:
T["cot"]    := { "class" = "Trigonometric" }:
T["coth"]   := { "class" = "Hyperbolic Trig" }:
T["csc"]    := { "class" = "Trigonometric" }:
T["csch"]   := { "class" = "Hyperbolic Trig" }:

T["currentdir"] := {
                "ES" = {"r" = 1..1, 0 = 0..1},
                "ST" = {"r" = {MIF(STRING)},
                          1  = TA("name/string")},
                "IO" = true }:

# ST_imp (diff of rational)
T["ldegree"] := { "ES" = {"r" = 1..1, 0 = 1..2},
                "ST" = {"r" = {MIF(NAME),MIF(PROD),
                               MIF(INTPOS),MIF(INTNEG)},
                          1  = TA("Expression"),
                          1  = TA("Expression")},
                    "Pure" = true };
T["degree"] := { "ES" = {"r" = 1..1, 0 = 1..2},
                  "ST" = {"r" = {MIF(NAME),MIF(PROD),
                                  MIF(INTPOS),MIF(INTNEG)},
                           1   = TA("Expression"),
                           1   = TA("Expression")},

                  "Pure" = true };
T["diff"]   := { "ES" = {"r" = 1..1, 0 = 2..infinity},
                  "Pure" = true };

T["entries"]    := { "ES" = {"r" = 0..infinity, 0 = 1..1},
                      "ST" = {"r" = {MIF(LIST)},
                               1   = {MIF(TABLE)}
                                      union TA("Name")},
                      "Pure" = true, "Builtin" = true };

T["eval"]    := { "ES" = {"r" = 1..1, 0 = 1..2},
                   "Pure" = true, "Builtin" = true };
T["evalb"]   := { "ES" = {"r" = 1..1, 0 = 1..1},
                   "ST" = { 1   = TA("Boolean")
                                  union TA("Assignable")},
                  "Pure" = true, "Builtin" = true };
T["evalf"]   := { "ES" = {"r" = 1..1, 0 = 1..2},
                   "ST" = {"r" = {MIF(COMPLEX)}
                                  union TA("Assignable")
                             union TA("float"),
                             2   = {MIF(INTPOS)}},
                   "Pure" = true, "Builtin" = true };
T["exp"]     := { "class" = "Elementary" };

T["exports"]:= { "ES" = {"r" = 0..infinity, 0 = 1..2},
                  "ST" = {"r" = TA("Name"),
                           1   = TA("Name")
                                  union {MIF(MODULE)},
                           2   = TA("global")},
                  "LV" = { 2   = {_Inert_NAME("instance"),
                                   _Inert_NAME("typed")}},
                  "Pure" = true, "Builtin" = true };
```

```
T["factorial"] := { "ES" = {"r" = 1..1, 0 = 1..1},
                    "Pure" = true, "Builtin" = true }:

T["fclose"] := { "ES" = {"r" = 0..0, 0 = 1..infinity},
                 "IO" = true }:

T["floor"]  := { "ES" = {"r" = 1..1, 0 = 1..2},
                 "ST" = {"r" = {MIF(FUNCTION)}
                              union TA("integer")},
                 "Pure" = true, "Builtin" = true }:

T["fopen"]  := { "ES" = {"r" = 1..1, 0 = 2..3},
                 "ST" = {"r" = {MIF(INTPOS)},
                         1   = TA("name/string"),
                         2   = {MIF(NAME)},
                         3   = {MIF(NAME)}},
                 "IO" = true }:

T["fprintf"]  := { "ES" = {"r" = 1..1, 0 = 2..infinity},
                   "ST" = {"r" = {MIF(INTPOS)},
                           2   = TA("name/string") },
                   "IO" = true }:

T["FromInert"] := { "ES" = {"r" = 0..infinity, 0 = 1..1},
                    "ST" = {"r" = TA("AnyValue"),
                            1   = {MIF(FUNCTION)},
                            2   = {MIF(EQUATION)},
                            3   = {MIF(EQUATION)}},
                    "Pure" = true, "Builtin" = true }:

T["getenv"] := { "ES" = {"r" = 1..1, 0 = 1..1},
                 "ST" = {"r" = {MIF(STRING)},
                         1   = TA("name/string")},
                 "IO" = true }:

T["has"]    := { "ES" = {"r" = 1..1, 0 = 2..2},
                 "ST" = {"r" = {_Inert_NAME}}, # truefalse
                 "LV" = {"r" = {_Inert_NAME("true"),
                                 _Inert_NAME("false")}},
                 "Pure" = true, "Builtin" = true }:
```

```
T["if"]      := { "ES" = {"r" = 0..infinity, 0 = 3..3},
                  "ST" = { 1  = {_Inert_NAME}},
                  "LV" = { 1  = {_Inert_NAME("true"),
                                 _Inert_NAME("false"),
                                 _Inert_NAME("FAIL")}},
                  "Pure" = true, "Builtin" = true }:
T["igcd"]    := { "ES" = {"r" = 1..1, 0 = 0..infinity},
                  "ST" = {"r" = {MIF(FUNCTION)}
                                union TA("integer")},
                  "Pure" = true, "Builtin" = true }:
T["ilog" ]   := { "ES" = {"r" = 1..1, 0 = 1..1},
                  "ST" = {"r" = {MIF(FUNCTION)}
                                union TA("integer")},
                  "Pure" = true, "Builtin" = true }:
T["ilog2"]   := { "ES" = {"r" = 1..1, 0 = 1..1},
                  "ST" = {"r" = {MIF(FUNCTION)}
                                union TA("integer")},
                  "Pure" = true, "Builtin" = true }:
T["ilcm"]    := { "ES" = {"r" = 1..1, 0 = 0..infinity},
                  "ST" = {"r" = {MIF(FUNCTION)}
                                union TA("integer")},
                  "Pure" = true, "Builtin" = true }:

T["iolib"]   := { "ES" = {"r" = 0..infinity, 0 = 1..infinity},
                  "ST" = { 1  = {MIF(INTPOS)}},
                  "IO" = true, "Pure" = false, "Builtin" = true }:

T["implies"]:= { "ES" = {"r" = 1..1, 0 = 2..2},
                  "ST" = {"r" = TA("Assignable")
                                union TA("Boolean"),
                           1  = TA("Assignable")
                                union TA("Boolean"),
                           2  = TA("Assignable")
                                union TA("Boolean")},
                  "class" = "logical" }:

T["isqrt"]   := { "ES" = {"r" = 1..1, 0 = 1..1},
                  "ST" = {"r" = {MIF(FUNCTION)} union TA("integer"),
                           1  = {MIF(NAME)} union TA("integer")},
                  "Pure" = true, "Builtin" = true }:
```

```
T["Im"]       := { "ES" = {"r" = 1..1,  0 = 1..1},
                   "Pure" = true, "Builtin" = true }:

T["indets"]  := { "ES" = {"r" = 1..1, 0 = 1..2},
                   "ST" = {"r" = {MIF(SET)}},
                   "Pure" = true, "Builtin" = true }:

T["indices"]   := { "ES" = {"r" = 0..infinity, 0 = 1..1},
                      "ST" = {"r" = {MIF(LIST)},
                               1  = {MIF(TABLE)} union TA("Name")},
                      "Pure" = true, "Builtin" = true }:

T["iquo"]     := { "ES" = {"r" = 1..1, 0 = 2..3},
                   "ST" = {"r" = {MIF(FUNCTION)} union TA("integer"),
                            1  = TA("Assignable") union TA("integer"),
                            2  = TA("Assignable") union TA("integer"),
                            3  = TA("Name")},
                   "Pure" = false, "Builtin" = true }:
T["irem"]     := { "ES" = {"r" = 1..1, 0 = 2..3},
                   "ST" = {"r" = {MIF(FUNCTION)} union TA("integer"),
                            1  = TA("Assignable") union TA("integer"),
                            2  = TA("Assignable") union TA("integer"),
                            3  = TA("Name")},
                   "Pure" = false, "Builtin" = true }:

T["int"]      := { "ES" = {"r" = 0..infinity, 0 = 2..infinity},
                   "ST" = { 2  = {MIF(EQUATION)} union TA("Name")},
                   "Pure" = false, "Builtin" = false }:

T["intersect"] := { "ES" = {"r" = 1..1, 0 = 1..infinity},
                      "ST" = {"r" = {MIF(SET)}
                                    union TA("Assignable") },
                      "Pure" = true }:

T["isprime"]:= { "ES" = {"r" = 1..1, 0 = 1..1},
                   "ST" = {"r" = {MIF(NAME),MIF(FUNCTION)},
                            1  = {MIF(INTPOS)}
                                 union TA("Assignable")},
                   "Pure" = true }:
```

```
T["kernel/transpose"] := { "ES" = {"r" = 1..1, 0 = 1..1},
                            "ST" = {"r" = {MIF(LIST)},
                                    1 = {MIF(LIST)}},
                      "Pure" = true, "Builtin" = true }:
T["kernelopts"] := { "ES" = {"r" = 0..infinity, 0 = 0..infinity},
                      "Pure" = true, "Builtin" = true }:
T["ln"]       := { "class" = "Elementary" }:
T["log"]      := { "class" = "Elementary" }:
T["log10"]    := { "class" = "Elementary" }:

T["lhs"]      := { "ES" = {"r" = 0..infinity, 0 = 1..1},
                   "ST" = { 1  = TA("relation") union {MIF(RANGE)}},
                   "Pure" = true, "Builtin" = true }:

T["lprint"]   := { "ES" = {"r" = 0..0, 0 = 0..infinity} }:

#ST_imp
T["map"]      := { "ES" = {"r" = 0..infinity, 0 = 2..infinity},
                   "Pure" = true, "Builtin" = true }:

T["Matrix"]   := { "ES" = { "r" = 1..1 },
                   "ST" = { "r" = {MIF(MATRIX)} },
                   "Pure" = true }:

T["max"]      := { "ES" = {"r" = 1..1, 0 = 0..infinity},
                   "Pure" = true, "Builtin" = true }:
T["member"]   := { "ES" = {"r" = 1..1, 0 = 2..3},
                   "ST" = {"r" = {MIF(NAME)}} }:

T["min"]      := { "ES" = {"r" = 1..1, 0 = 0..infinity},
                   "Pure" = true, "Builtin" = true }:

T["minus"]    := { "ES" = {"r" = 1..1, 0 = 2..2},
                   "ST" = {"r" = {_Inert_SET}
                              union TA("Assignable"),
                           1  = {_Inert_SET}
                              union TA("Assignable"),
                           2  = {_Inert_SET}
                              union TA("Assignable")},
                     ”Pure” = true , ”Builtin” = true };

T[”mkdir”]   := { ”ES” = {”r” = 0..0, 0 = 1..1},
                  ”ST” = { 1 = TA(”name/string”)},
                  ”IO” = true };

T[”modp”]    := { ”ES” = {”r” = 1..1, 0 = 2..2},
                  ”ST” = {”r” = TA(”integer”)
                               union TA(”Assignable”)},
                  ”Pure” = true , ”Builtin” = true };

T[”mods”]    := { ”ES” = {”r” = 1..1, 0 = 2..2},
                  ”ST” = {”r” = TA(”integer”)
                               union TA(”Assignable”)},
                  ”Pure” = true , ”Builtin” = true };

T[”mul”]     := { ”ES” = {”r” = 0..infinity , 0 = 2..2},
                  ”ST” = {”r” = TA(”Algebraic”),
                           2  = {MIF(EQUATION),MIF(FUNCTION)}} };

T[”nops”]    := { ”ES” = {”r” = 1..1, 0 = 1..1},
                  ”ST” = {”r” = {MIF(INTPOS)}},
                  ”Pure” = true , ”Builtin” = true };

T[”not”]     := { ”ES” = {”r” = 1..1, 0 = 1..1},
                  ”ST” = {”r” = TA(”Assignable”)
                               union TA(”Boolean”),
                           1  = TA(”Assignable”)
                               union TA(”Boolean”)},
                  ”class” = ”logical” };
T[”nprintf”]   := { ”ES” = {”r” = 1..1, 0 = 1..infinity},
                    ”ST” = {”r” = TA(”global”),
                             1  = TA(”name/string”) },
                    ”IO” = false };

T[”op”]      := { ”ES” = {”r” = 0..infinity , 0 = 1..2},
                  ”ST” = {”r” = TA(”AnyValue”)},
                  ”Pure” = true , ”Builtin” = true };

T[”or”]      := { ”ES” = {”r” = 1..1, 0 = 2..2},
                      "ST" = {"r" = TA("Assignable")
                                    union TA("Boolean"),
                               1  = TA("Assignable")
                                    union TA("Boolean"),
                               2  = TA("Assignable")
                                    union TA("Boolean")},
                      "class" = "logical" }:

T["parse"]   := { "ES" = {"r" = 0..infinity, 0 = 1..1},
                  "ST" = { 1  =  TA("name/string") },
                  "Pure" = false, "Builtin" = true }:

T["print"]   := { "ES" = {"r" = 0..0, 0 = 0..infinity} }:
T["printf"]  := { "ES" = {"r" = 0..0, 0 = 1..infinity},
                  "ST" = { 1  = TA("name/string") },
                  "IO" = true }:
T["Re"]      := { "ES" = {"r" = 1..1, 0 = 1..1},
                  "Pure" = true, "Builtin" = true }:
T["remove"]  := { "ES" = {"r" = 0..1, 0 = 2..infinity},
                  "ST" = {"r" = TA("AnyValue")},
                  "Pure" = true, "Builtin" = true }:

T["rmdir"]   := { "ES" = {"r" = 0..0, 0 = 1..1},
                  "ST" = { 1 = TA("name/string")},
                  "IO" = true }:

T["round"]   := { "ES" = {"r"=1..1, 0 = 1..2},
                  "ST" = {"r"={MIF(FUNCTION)} union TA("integer")},
                  "Pure" = true, "Builtin" = true }:
T["rhs"]     := { "ES" = {"r" = 0..infinity, 0 = 1..1},
                  "ST" = { 1  = TA("relation") union {MIF(RANGE)}},
                  "Pure" = true, "Builtin" = true }:

T["rtable"]   := { "ES" = {"r" = 1..1,  0 = 0..infinity},
                   "ST" = {"r" = {MIF(ARRAY)}},
                   "Pure" = true, "Builtin" = true }:
T["rtable_dims"] := {
                   "ES" = {"r" = 0..infinity, 0 = 1..infinity},
                   "ST" = {"r" = {MIF(RANGE),MIF(EXPSEQ)},
                            1  = TA("Name") union {MIF(ARRAY)}},
                   "Pure" = true, "Builtin" = true }:
T["rtable_elems"] := {
                   "ES" = {"r" = 1..1, 0 = 1..infinity},
                   "ST" = {"r" = {MIF(SET)},
                            1  = TA("Name") union {MIF(ARRAY)}},
                   "Pure" = true, "Builtin" = true }:
T["rtable_num_dims"] := {
                   "ES" = {"r" = 1..1, 0 = 1..infinity},
                   "ST" = {"r" = {MIF(INTPOS)},
                            1  = TA("Name") union {MIF(ARRAY)}},
                   "Pure" = true, "Builtin" = true }:
T["rtable_num_elems"] := {
                   "ES" = {"r" = 1..1, 0 = 1..2},
                   "ST" = {"r" = {MIF(INTPOS)},
                            1  = TA("Name") union {MIF(ARRAY)}},
                   "Pure" = true, "Builtin" = true }:

T["searchtext"] := { "ES" = {"r" = 1..1, 0 = 2..3},
                     "ST" = {"r" = {MIF(INTPOS)},
                              1  = TA("name/string"),
                              2  = TA("name/string"),
                              3  = {MIF(RANGE)}},
                     "Pure" = true, "Builtin" = true }:
T["SearchText"] := { "ES" = {"r" = 1..1, 0 = 2..3},
                     "ST" = {"r" = {MIF(INTPOS)},
                              1  = TA("name/string"),
                              2  = TA("name/string"),
                              3  = {MIF(RANGE)}},
                     "Pure" = true, "Builtin" = true }:
T["sec"]     := { "class" = "Trigonometric" }:
T["sech"]    := { "class" = "Hyperbolic Trig" }:
T["select"]  := { "ES" = {"r" = 0..1, 0 = 2..infinity},
                  "Pure" = true, "Builtin" = true }:
T["selectremove"] := { "ES" = {"r" = 1..2, 0 = 2..infinity},
                       "Pure" = true, "Builtin" = true }:
T["seq"]       := { "ES" = {"r" = 0..infinity, 0 = 1..3},
                    "Pure" = false, "Builtin" = true }:
T["series"]    := { "ES" = {"r" = 1..1, 0 = 2..3},
                    "ST" = {"r" = {MIF(SERIES),MIF(EXACTSERIES)},
                             2  = TA("Name") union {MIF(EQUATION)},
                                  3   = {MIF(INTPOS)}},
                        "Pure" = false, "Builtin" = true }:
T["setattribute"] := { "ES" = {"r" = 1..1, 0 = 1..infinity},
                          "Pure" = true, "Builtin" = true }:
T["sign"]        := { "ES" = {"r" = 1..1, 0 = 1..3},
                        "ST" = {"r" = TA("integer"),
                                  2   = {MIF(LIST)},
                                  3   = TA("Name")},
                        "LV" = {"r" = {_Inert_INTPOS(1),
                                        _Inert_INTNEG(1)}},
                        "Pure" = false, "Builtin" = true }:
T["signum"]      := { "ES" = {"r" = 1..1, 0 = 1..3},
                        "Pure" = false, "Builtin" = true }:
T["sin"]         := { "class" = "Trigonometric" }:
T["sinh"]        := { "class" = "Hyperbolic Trig" }:
T["sprintf"]     := { "ES" = {"r" = 1..1, 0 = 1..infinity},
                        "ST" = {"r" = {MIF(STRING)},
                                  1   = TA("name/string") },
                        "IO" = false }:
T["ssystem"]     := { "ES" = {"r" = 1..1, 0 = 1..1},
                        "ST" = {"r" = {MIF(STRING)},
                                  1   = TA("name/string")},
                        "IO" = true, "Pure" = false, "Builtin" = true }:
T["subs"]        := { "ES" = {"r" = 1..1, 0 = 1..infinity},
                        "Pure" = true, "Builtin" = true }:
T["subset"]      := { "ES" = {"r" = 1..1, 0 = 2..2},
                        "ST" = {"r" = {MIF(NAME),MIF(FUNCTION)},
                                  1   = {MIF(SET)} union TA("Assignable"),
                                  2   = {MIF(SET)} union TA("Assignable")},
                        "Pure" = true, "Builtin" = true }:
T["subsop"]      := { "ES" = {"r" = 1..1, 0 = 1..infinity},
                        "Pure" = true, "Builtin" = true }:
T["subtype"]     := { "ES" = {"r" = 1..1, 0 = 2..2},
                        "ST" = {"r" = {MIF(NAME)}},
                        "LV" = { "r" = {_Inert_NAME("true"),
                                        _Inert_NAME("false"),
                                        _Inert_NAME("FAIL")}},
                        "Pure" = true, "Builtin" = false }:
T["substring"]:= { "ES" = {"r" = 1..1, 0 = 2..2},
                        "ST" = {"r" = TA("name/string"),
                                   1   = TA("name/string"),
                                   2   = {MIF(RANGE)} union
                                         TA("integer")},
                         "Pure" = true, "Builtin" = true }:
T["sum"]      := { "ES" = {"r" = 0..infinity, 0 = 2..2},
                   "ST" = {"r" = TA("Algebraic"),
                           2   = TA("Name") union {MIF(EQUATION)}},
                   "Pure" = false, "Builtin" = false }:

T["symmdiff"] := { "ES" = {"r" = 1..1, 0 = 0..infinity},
                   "ST" = {"r" = {MIF(SET)} union
                                 TA("Assignable") },
                   "Pure" = true }:

T["system"]   := { "ES" = {"r" = 1..1, 0 = 1..1},
                   "ST" = {"r" = {MIF(INTPOS)},
                           1   = TA("name/string")},
                   "IO" = true, "Pure" = false, "Builtin" = true }:

T["table"]    := { "ES" = {"r" = 1..1, 0 = 1..2},
                   "ST" = {"r" = {MIF(TABLE)},
                           2   = TA("sequential")},
                   "Pure" = true, "Builtin" = true }:

T["tan"]        := { "class" = "Trigonometric" }:
T["tanh"]       := { "class" = "Hyperbolic Trig" }:
T["taylor"]     := { "ES" = {"r" = 1..1, 0 = 2..3},
                     "ST" = {"r" = {MIF(SERIES),MIF(EXACTSERIES)},
                             2   = {MIF(EQUATION)},
                             3   = {MIF(NAME),MIF(INTPOS)}},
                     "Pure" = false, "Builtin" = true }:
T["ToInert"]    := { "ES" = {"r" = 0..infinity, 0 = 0..infinity},
                     "ST" = {"r" = {MIF(FUNCTION)}},
                     "Pure" = true, "Builtin" = true }:

T["trunc"]    := { "ES" = {"r" = 1..1, 0 = 1..2},
                   "ST" = {"r" = {MIF(FUNCTION)} union TA("integer")},
                   "Pure" = true, "Builtin" = true }:

T["type"]     := { "ES" = {"r" = 1..1, 0 = 1..2},
                         "ST" = {"r" = {MIF(NAME)}}, # truefalse
                         "LV" = { "r" = {_Inert_NAME("true"),
                                         _Inert_NAME("false")}},
                         "Pure" = true, "Builtin" = true }:

    T["typematch"] := {
                         "ES" = {"r" = 1..1, 0 = 1..2},
                         "ST" = {"r" = {MIF(NAME)}}, # truefalse
                         "LV" = {"r" = {_Inert_NAME("true"),
                                        _Inert_NAME("false")}},
                         "Pure" = false, "Builtin" = true }:

    T["union"]   := { "ES" = {"r" = 1..1, 0 = 0..infinity},
                         "ST" = {"r" = {MIF(SET), MIF(FUNCTION)}},
                         "Pure" = true, "Builtin" = true }:

    T["userinfo"]:= { "ES" = {"r" = 0..0, 0 = 3..infinity },
                          "ST" = { 1  = {MIF(INTPOS)} },
                          "Pure" = true, "Builtin" = true }:

    T["vector"] := { "ES" = { "r" = 1..1, 0=1..infinity },
                         "ST" = { "r" = {MIF(TABLE)} },
                         "Pure" = true }:
    T["Vector"] := { "ST" = { "r" = TA("Vector") },
                         "Pure" = true }:

    T["xor"]   := { "ES" = {"r" = 1..1, 0 = 2..2},
                       "ST" = {"r" = TA("Assignable") union TA("Boolean"),
                               1  = TA("Assignable") union TA("Boolean"),
                               2  = TA("Assignable") union TA("Boolean")},
                         "class" = "logical" }:
end module:
```

Bibliography

[1] Luca Cardelli. A polymorphic λ-calculus with Type:Type. Technical Report 10, DECSRC, May 1986.

[2] J. Carette and S. Forrest. Mining Maple code for contracts. In Ranise and Bigatti [25].

[3] J. Carette and S. Forrest. Property inference for Maple. Technical report, University of Linz, Austria, 2007. in RISC Technical Report 07–06.

[4] J. Carette and M. Kucera. Partial Evaluation for Maple. In *ACM SIGPLAN 2007 Workshop on Partial Evaluation and Program Manipulation*, 2007.

[5] P. Cousot. Program analysis: The abstract interpretation perspective. *ACM Computing Surveys*, 28A(4es):165–es, December 1996.

[6] P. Cousot. Types as abstract interpretations, invited paper. In *Conference Record of the Twentyfourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 316–331, Paris, France, January 1997. ACM Press, New York, NY.

[7] P. Cousot. A tutorial on abstract interpretation. In *VMCAI'05 Industrial day on Automatic Tools for Program Verification*, Maison des Polytechniciens, Paris, France, January 20 2005.

[8] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[9] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[10] P. Cousot and R. Cousot. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics*, 81(1):43–57, 1979.

[11] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[12] P. Cousot and R. Cousot. Comparison of the Galois connection and widening/narrowing approaches to abstract interpretation. JTASPEFL '91, Bordeaux. *BIGRE*, 74:107–110, October 1991.

[13] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '92,*, Leuven, Belgium, 13–17 August 1992, Lecture Notes in Computer Science 631, pages 269–295. Springer-Verlag, Berlin, Germany, 1992.

[14] P. Cousot and R. Cousot. Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form, invited paper. In P. Wolper, editor, *Proceedings of the Seventh International Conference on Computer Aided Verification, CAV '95*, pages 293–308, Liège, Belgium, Lecture Notes in Computer Science 939, 3–5 July 1995. Springer-Verlag, Berlin, Germany.

[15] P. Cousot and R. Cousot. *Basic Concepts of Abstract Interpretation*, pages 359–366. Kluwer Academic Publishers, 2004.

[16] Patrick Cousot. Abstract interpretation. MIT course 16.399, `http://web.mit.edu/16.399/www/`, Feb.–May 2005.

[17] Brian A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, 2002.

[18] P. DeMarco, K. Geddes, K. M. Heal, G. Labahn, J. McCarron, M. B. Monagan, and S. M. Vorkoetter. *Maple 10 Advanced Programming Guide*. Maplesoft, 2005.

[19] George Grätzer. *General Lattice Theory*. Birkhäuser Verlag, Basel und Stuttgart, 1978.

[20] M. Kucera and J. Carette. Partial evaluation and residual theorems in computer algebra. In Ranise and Bigatti [25].

[21] Antoine Miné. Representation of two-variable difference or sum constraint sets and application to automatic program analysis. Master's thesis, Laboratoire d'Informatique de l'ENS, 2000.

[22] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[23] S. L. Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, April 2003.

[24] Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

[25] Silvio Ranise and Anna Bigatti, editors. *Proceedings of Calculemus 2006*, Electronic Notes in Theoretical Computer Science. Elsevier, 2006.

[26] Mads Rosendahl. Introduction to abstract interpretation. DIKU, Computer Science, University of Copenhagen, 1995.

[27] David Schmidt. Abstract interpretation and static analysis. Lectures at the Winter School on Semantics and Applications, WSSA'03, Montevideo, Uruguay., July 2003.

[28] Michael Schwartzbach. Lecture notes in static analysis. Basic Research in Computer Science, University of Aarhus, Denmark.

[29] Michel Sintzoff. Calculating properties of programs by valuations on specific models. In *Proceedings of ACM conference on Proving assertions about programs*, pages 203–207, New York, NY, USA, 1972. ACM Press.

Index

PROPERTY INFERENCE FOR MAPLE