Mining Maple Code for Contracts

Jacques Carette [1] Stephen Forrest [2]

Computing and Software
McMaster University
Hamilton, Ontario, Canada

Abstract

We wish to answer the following question: what is the most appropriate language for describing the "contracts" that Maple routines offer? In this, we are seeking much more than types (which Maple does not have, at least statically), as these are not sufficiently expressive to capture what is going on. We also wish to study what is actually in Maple, rather than what *should be there*. Put another way, we do not expect to find that a type system like Aldor's or Axiom's would be especially helpful in explaining Maple. Our real goal is a mathematical description of the interfaces between routines. As such, the only current terminology flexible enough to encompass reality is that of contracts, by which we mean simply statements of complex properties (static as well as dynamic) in a sufficiently general logic.

This works focuses mainly on the requirements analysis phase of this project: we perform automated analyses of the complete Maple library, to understand the kinds of contracts that are in actual use. We wish to know which kinds of theorems would need to be proved in order to formally analyze the types, effects, invariants, and contracts present in the current code base. As this is a monumental task, we describe what knowledge we have currently been able to extract from a very systematic approach to the problem.

1 Introduction

Our initial goal was to write a prototype type inferencer for (parts of) Maple. When this led to certain difficulties, it seemed that we should instead employ a "type-and-effects" system [6]. Ultimately, this too seemed unfruitful, and a re-evaluation of our approach was necessary. Analysis of the obstacles faced led us to conclude that what we had expected to find (which drove our initial design) diverged considerably from what was actually present in *typical* library code.

The current work grew out of this impasse. We wanted to begin anew, and get a realistic view of what is actually in Maple's library as evidenced by its source code.

[1] carette@mcmaster.ca
[2] forressa@mcmaster.ca

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

While we had originally thought that a type system was a good means of "inferring" this information, it became clear that standard type systems based on Hindley-Milner type inference and System F [8,9] would not be sufficient. Re-examining our original goals, we saw that code comprehension was more important to us than any particular formalism. This was clearly a sign that, in time-honoured mathematical tradition, we needed to widen rather than narrow our scope. Some experience with the programming language Haskell [7] led us to speculate that *type classes* could help. Also influential were Eiffel [5] and various specification mechanisms (most notably algebraic specifications with a categorical flavour following Goguen [2]). However, since we did not know exactly what we were seeking, we settled upon a rather vague term, *contract*, that we hoped would be general enough to encompass our needs. Contracts are very familiar to the specification community (but under the terms used by Floyd, Hoare, and Dijkstra, that is pre and post-conditions as well as invariants), and increasingly to the object-oriented community as well [4]. They are also making some inroads into the functional programming world [3]. However, we do not wish to use this word too formally **yet**, and certainly our use differs from that of [4,3].

We also knew that a weak logic (like first-order logic) would be insufficient to handle Maple's higher-order functions. Indeed, even higher-order logic is not sufficient, since Maple has both reflection and reification (e.g. explicitly delayed evaluation, `eval`, `ToInert` and `FromInert`, `pointto` and `addressof`, etc), first-class "types", dependent "types", and so on. (For more information on some Maple-specific language constructs, refer to our Appendix.A) In view of these advanced features and of previous work [10], it seemed clear that even a higher-order modal logic may not even fit. So we embarked upon the road of *requirements analysis*: what *language* could we use to express the structures found in Maple code?

We therefore decided to use Maple's own library as a large unstructured database, and mine it for requirements. This paper reflects our current understanding of what our tools have unearthed, and what requirements we have been able to extract from these results.

As this could easily devolve into an *ad hoc* foraging expedition, we needed to settle on a reasonable methodology by which we could systematically investigate the data at hand. Section 2 explains our methodology, our tool and its evolution, while section 3 gives a description of the stages of the tool. Section 4 gives an overview of our main data-mining results. Section 5 presents a discussion of interesting nuggets of information we have been able to dig out of this same data, and some explicit examples of our results. Finally, in section 6, we give the main requirements we have been able to gather until now, and offer an overview of the directions in which we wish to proceed further. In appendix A we provide a glossary which defines some of the more arcane Maple terminology that we use; unfortunately space requirements mean that some familiarity with Maple is needed to comfortably read this work.

2 Methodology

Both the authors know rather "too much" about Maple and its innards. This knowledge could seriously handicap an experimental effort to discover the *actual* state of the codebase, as the seeds of knowledge "discovered" by the inference engine may only be those planted by its authors' implicit bias! To counter this, the first methodological decision was to use a naïve process. More specifically, we wish to build in to our process no *a priori* knowledge, but to construct an inference engine based only on what is found at each turn.

To this end, we first "exploded" the complete Maple library into a form more suitable for automatic manipulation. Maple's `ToInert` function, coupled with the tools from `LibraryTools` are exactly what we needed: with these we could extract an accurate Abstract Syntax representation of every global name stored in Maple's library. Maple's internal representation (seen through the reflection tools) is essentially like LISP s-expressions: a header indicating the low-level "type" of the object, and an ordered sequence of sub-objects. In Maple notation, this looks like `Header(sub1, sub2, sub3, sub4)`, where each sub-object is similar; the leaf nodes are either literal integers or literal strings. More details can be found in Appendix A and [1].

The methodology we followed is perhaps best described as *opportunistic*. We sought to find as much as possible, as quickly as possible. Two natural approaches suggested themselves: a *top-down* and a *bottom-up* approach.
The top-down approach would proceed in this manner:

1. Construct a list of items as yet unhandled by the inferencer, ordered by decreasing number of occurrences,
2. Implement a simple approach to dealing with the first item in the above list,
3. Go back to 1.

This ensures maximal coverage of the most important parts of the library as early as possible. One disadvantage is that very simple things might never occur frequently enough to be handled. For example, if there were but 20 raw strings in the library, these might not be "typed" for a very long time, even though this would be easy to do.
The bottom-up approach would proceed in the following manner:

1. Construct a list of items as yet unhandled by the inferencer, ordered by increasing complexity,
2. Implement a simple approach to dealing with the first item in the above list,
3. Go back to 1.

This might also be called a greedy approach, one in which the simplest parts would be done first, and results would arrive early. A drawback is that some large pieces of code that are nevertheless "simple" remain unhandled and even unexamined for a very long time.

The approach we ultimately took was a somewhat idiosyncratic combination of the these two. We first do a top-down step, then a bottom-up step, and repeat. As well, because computing a reasonable "complexity" of Maple objects is

not straightforward, we decided that the bottom-up steps would be done manually, somewhat at the whim of the implementor.

Consequently, while we are quite interested in a very systematic approach to this problem, we decided to take this rather more pragmatic approach to maximize the results we could get. Nevertheless, this hybrid strategy still preserves the *a priori* nature of our approach, as our results are merely a refinement of those produced by a purely top-down approach.

The methodology above clearly defines a notion of *stages*, corresponding to new "features" of the inferencer, depending on what each phase above says should be done next. We felt that it would be very important to be able to re-run our inferencer at any given stage, especially since a bug in one stage could completely throw off our histograms. Thus we built our entire system with a *history* mechanism. In other words, at run-time, the inferencer is first initialized to work at a particular stage, and then the inference proceeds on the whole library. This particular decision required a much more complex system architecture, but we also feel that it was a key factor in being able to push things as far as we did.

Also key to our methodology was our desire to heavily bias our system towards what was actually *found* in the code, rather than what we might have expected or what perhaps *should* be present in software which implements mathematics. Our past experience told us that while the structures of mathematics were implicitly present in a significant portion of the code, they were not in fact so easy to find explicitly. For example, although one would expect to find many Monoids in code handling algebraic mathematics, in practice they seem to either be rarer than one would think, or perhaps more thoroughly disguised.

As mentioned in the introduction, we at first wanted to write a system which would serve as the main **requirements analysis** tool for a pure *type inference* engine for Maple. On top of our pre-existing familiarity with Maple (and its underlying mathematics), we were also influenced by the types of two different systems: Haskell [7] and Aldor [11]. However, we knew that this would not be sufficient since Maple supports reflection and reification, first-class "types", dependent "types", etc. We opted instead to look for any kind of logical conclusion we could draw from the code. From a formal methods perspective, it became clear that we were looking at were *relations* satisfied between the input of a procedure, the output of the procedure, and the environment, as an approximation to an axiomatic description of the operational semantics of the library code. However, the word *contract* seemed to convey what we were after more succinctly.

What are we trying to find? Consider the following Maple procedure:

```
proc(a,t,x)
    map(Int, indets(a, t), x=0..1)
end proc
```

The code's size belies its complexity. It returns a set of unevaluated definite integrals with respect to `x` over the interval $[0,1]$, where the integrands are all the subexpressions matching type `t` which are present in `a`.

We would be happy to have a system that could tell us that `a` must be a *traversable* object, `t` must represent a Maple *type*, `x` must be a *symbol*, and that the result must

be a `set` of objects of type `Int(t, x=0..1)`. Additionally, we should like to know that this is a pure function (i.e. no side-effects), that all 3 arguments are used but additional arguments, if given, are simply ignored. We could also demand that the results *make sense*, in which case we would further constrain `t` to represent a type of integrable expressions. Note that this would not constrain `a`, as it would still be sensible to use this procedure when we knew that `a` contained no sub-objects of type `t`. Though our actual representation of these results is more complex, we could write a pretty-printer for the types that could output a representation like

$$(\texttt{Traversable}(a), \texttt{Type}(t), \texttt{Symbol}(x)) \Rightarrow \\ (a, t, x, \$\_) \rightarrow \{\underline{\texttt{Int}}((b : t), x \underline{=} \underline{0..1})\} \Leftarrow b \preccurlyeq a$$

where constraints on the input types appear before the $\Rightarrow$, $\$\_$ delimits the variable arguments, constraints on the output types appear after the $\Leftarrow$, underlines (e.g. $\underline{\texttt{Int}}$) indicate value embeddings at the type level, and $\preccurlyeq$ is the sub-object relation (at the type level). The above ought to be taken as an indication of our *intentions*, rather than as an example from a formal contract system.

3 System overview

In this section, we give an overview of our process. To interpret the results, it is crucial to comprehend the multi-stage inferencer as described in the previous section.

Below, we describe each stage and explain what new "feature" the stage implements. There are currently 53 stages; this stopping-point is quite arbitrary, and was chosen merely because it was time to write up our results rather than because we had finished the task or arrived at a "natural" endpoint.

While these 53 stages were certainly sufficient to obtain many interesting results, we believe that at least 100 stages or more would be necessary to achieve reasonable coverage of the full Maple library. Nevertheless, we feel that this process has already uncovered sufficiently many requirements for a contract system to demonstrate the value of our approach.

All of the items below are marked with either a **T**or a **B**, indicating respectively a top-down or bottom-up stage. It is also worth noting that we separate an analysis phase (recursing through the components of an object) and the reconstruction phase where we "build" the contract of the object as a whole from the sub-contracts. For reasons of length, we will assume that the reader is familiar with Maple's object representation; an appendix to the *Advanced Programming Guide* [1] provides a good introduction to these details. However, a detailed understanding is not necessary to comprehend the general concepts behind our approach.

We will follow the Maple naming convention of using all-capitals to name internal objects; these names correspond to those used in Maple's inert form A.

1 **T** On start-up, the only thing known is that the library consists of a lot (13063) of named objects, each of which is given by an Abstract Syntax Tree, with a

named root node and sub-trees. Each node in the tree has a "type" which corresponds to an object of the 37 visible Maple internal tags for its own internal representation. Table 1 gives the full results of this stage. We make one concession to *a priori* knowledge: all names that start with `type/` are treated as types, not values.

2 **B** Leaf objects are implemented. In other words, INTPOS, INTNEG, STRING, and FLOAT. RATIONAL and COMPLEX are implemented too.

3 **T** As procedures are (by far) the most common, recurse on each component.

4 **B** Recurse through SET, LIST, and EXPSEQ. Types for these can all be straightforwardly rebuilt from the components. In fact, if all sub-objects are static values, then it is reconstructed as a Static value.

5 **T** Stage 3 was completely naïve and, as expected, pushes the problem to the first component of procedures (the parameter sequence PARAMSEQ). We implement (again by simple recursion) traversal of PARAMSEQ, LOCALSEQ, OPTIONSEQ, LEXICALSEQ, RETURNTYPE. We give the global sequence (GLOBALSEQ) the type `None` as it has no operational significance.

6 **B** Recurse through EQUATION and POWER. Static cases handled.

7 **T** Again, Stage 5 handling was naïve, in particular with respect to locals. Create a recursive *environment* where bindings for locals (and parameters) can be stored, and deal with NAME and DCOLON entries in LOCALSEQ.

8 **B** Global NAMEs in the library are, by definition, *symbols*.

9 **T** Deal with unknown options. Options *remember*, *operator*, *arrow*, *system* and copyright strings have no operational meaning, so they are stripped out. Other options will be treated later.

10 **B** Recurse through NOT, UNEVAL, and FUNCTION (i.e. function applications). Handle Static cases of NOT and UNEVAL.

11 **T** Like stage 7, but for parameters.

12 **B** Recurse through PROD (and handle Static).

13 **T** We are finally at the stage where procedure's actual STATSEQ is the main blocking point! Right now, just recurse through, and accumulate information. The resulting value will be that of the last statement in the STATSEQ (until we are ready to deal with more complex control flow).

14 **B** Recurse through RANGE, CATENATE, SUM, AND, and OR, handling Static cases.

15 **T** Somewhat surprisingly, the most common object in a Maple procedure amongst all the "statements" is ASSIGN. We do not take this to mean that Maple is fundamentally imperative, as many assignments could be done via a *let* expression, if Maple had them. Right hand sides of assignments are typed, and a binding to that type is added to the left hand side in the environment.

16 **B** Recurse through INEQUAT, MEMBER, and NARGS, handling Static cases.

17 **T** Deal with ASSIGNEDNAME. This corresponds to the use of a name which either has a meaning in the library or is a builtin function. If the name has already been typed (successfully or otherwise), the result is taken from the environment. Recursion (through names) is not handled, and caught via the environment. Failures for each builtin is given by name, causing an explosion of the number of unhandled cases at all later stages.

18 **B** Handle function calls when the function called is a *symbol*.
19 **T** The second-most popular statement is a conditional. As an IF is always composed of at least one CONDPAIR, recurse through these too. Also, put a guard to ensure that conditions in an IF have type "boolean".
20 **B** Recurse through LESSTHAN and LESSEQ.
21 **T** Implement some inference for Maple types (InferType). All symbols are considered to be types at this stage.
22 **B** Handle a DCOLON. Ensure right hand side corresponds to a Maple type.
23 **T** Handle references to PARAM. The environment is queried for the type.
24 **B** Give a type to the builtins `evalb`, and `length` (as they are amongst the simplest builtins). `evalb` takes in any single Maple object and will return `true` or `false`, while `length` takes any single Maple object and returns a positive integer.
25 Due to some hiccup in our development, this stage number was skipped.
26 **T** Start giving types to non "Static" objects. For this stage, just implement dispatching to internal-type driven tables. We will call the infering of contracts from pieces "re-assembling".
27 **B** Re-assemble EQUATION, INEQUAT, LESSTHAN, LESSEQ as well as 1 and 2 argument EXPSEQ. 1 argument EXPSEQ occur for example in the inert representation of a list like $[a]$ where a is a parameter.
28 **T** First implementation of VerifyBoolean, the routine to check that an expected boolean really is.
29 **B** Recurse through XOR
30 **T** References to LOCAL are most common, and handled much like PARAM.
31 **B** Re-assemble AND, OR, and NOT.
32 **T** We give a contract for `type`. This takes any Maple object as first parameter, a Maple type as second, and returns a value of type `truefalse`.
33 **B** InferType support for SET, LIST, EQUATION, and RANGE. Since Maple types are quite different than values, even though they are implemented as values, it is important to infer them separately.
34 **T** Re-assemble ASSIGN. Amounts to returning the type of the right hand side.
35 **B** VerifyBoolean - add support for Static values, as well as the type constructors `Not`, `And` and `Or`.
36 **T** Re-assembling of FUNCTION (function calls). Make sure that when calling a function, that the types expected and the arguments passed are "compatible".
37 **B** Check if Static values are actually types. This occurs surprisingly often.
38 **T** Recurse through TABLEREF.
39 **B** Implement checks that $<$ and $\leq$ are booleans, which means verifying that the arguments are numeric. Implement VerifyNumeric for this task. Deal with Static values, and (Maple) subtypes of "numeric".
40 **T** Calls to builtin `op` are most common. Deal with this by creating a name for the type of the function `op`, call it *OpType*. Because `op` is ad-hoc polymorphic, this allows us to type `op` at the call site instead.
41 **B** Re-assemble (some) TABLEREF.

42 **T** Now *OpType* can be be implemented at the call site. Create what is essentially a type class, called *OpAble* to represent all objects which can be `op`'ed. *OpAble* tracks whether 1- or 2-argument `op` was used, and in the 2-argument case, what that was. This is done as a *constraint*.
43 **B** Static TABLEREF, re-assembly of SET, LIST.
44 **T** Type for builtin `nops`. As an approximation, use type *anything* for the input, although type *OpAble* might be more accurate.
45 **B** Previous handling of UNEVAL was naïve (it basically did nothing!). Now set an environment variable so that ASSIGNEDNAME encountered inside an UNEVAL are treated as a NAME.
46 **T** Handle ERROR statement. Get the type of the components, but then return *None*, which is the absence of a type. This does not correctly handle the case where there is code below an ERROR, but this should be very rare.
47 **B** InferType implemented over ASSIGNEDNAME, and UNEVAL.
48 **T** Implement ARGS. Like `op`, this is done by creating a new type *Sequence-Variable* (where clearly Mathematica has influenced our choice of name).
49 **B** Re-assemble DCOLON, UNEVAL and RANGE
50 **T** Implement RETURN. Like ERROR, the 'value' of a RETURN is actually *None*, however RETURN adds its value as a *result type* in the environment. The result of a procedure will be composed of the appropriate combination of all obtained result types.
51 **B** VerifyBoolean now know about `::`, and Parameter. When encountering a Parameter, it adds a constraint to the environment.
52 **T** Re-assemble simple IF. "simple" is defined to mean an IF with 2 branches and identical types in both branches, or a one branch IF of type *None*.
53 **B** InferType can now handle POWER, FUNCTION and RETURNTYPE.

We plan to pursue this line of study and record our detailed results in a technical report; also, we hope to eventually make our inferencer code available as we feel that it could in time be useful to others as well.

4 Results

Table 1 shows the results we get for stage 1, where for space reasons we use A* to designate the long name ASSIGNED. As expected, procedures make up the bulk of the library. The large number of tables is in part due to `inttrans` which uses a pattern-matching approach to computing integral transforms, and includes a large number of data tables. The *InferType* entry refers to the 409 `type/` routines. See the next section for a further discussion of these findings.

Figure 1 shows a plot of the number of named Maple objects which are successfully "typed" at each stage. The jump at stage 2 reflects the typing of 122 objects (the raw integers, floating point numbers, etc) in the library. The jump at stage 4 reflects the typing of 265 lists (of integers, floats, strings, 244 are lists of lists of either floats or integers) as well as 11 expression sequences and 11 sets (all empty!). Stage 8 reflects the typing of symbols (264 of them, as well as 32 compound objects containing symbols). Stage 16 reflects the typing of very simple procedures

Number	11023	457	409	297	264
Tag	PROC	TABLE	InferType	LIST	NAME
Number	164	141	84	71	55
Tag	MODULE	INTPOS	TABLEREF	SET	FUNCTION
Number	39	16	16	9	9
Tag	A*NAME	EXPSEQ	ARRAY	LOCALNAME	FLOAT
Number	3	3	1	1	1
Tag	STRING	UNEVAL	INTNEG	COMPLEX	A*LOCALNAME

Table 1
Results of first-stage failures

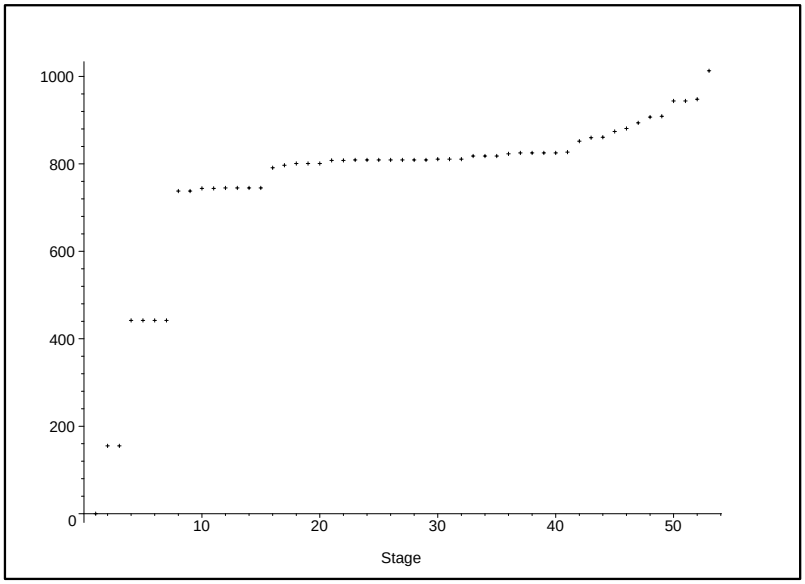

Fig. 1. Number of typed objects per stage

like `proc() 0 end proc` (there are 17 occurrences of this in the library). Further advances are then hard-won, and reflect the many features that have had to be implemented to get to that point, documented in the previous section.

Figure 2 shows, from stage 16 onwards, the difference in the number of successfully typed objects at each successive stage. Unlike the early successes shown in the previous figure, it is clear that few Maple procedures are completely trivial, and that a fair amount of infrastructure is necessary before steady progress occurs. However, one can see that, beginning at approximately stage 40, our efforts start to pay off, and successes come more readily.

Figure 3 displays, again from stage 16 onwards, the number of changed Failure messages at each stage. This reflects movement from one source of failure to another. What is most interesting is the overall even/odd separation: one can see a "high" curve corresponding to the large movements due to the top-down passes,

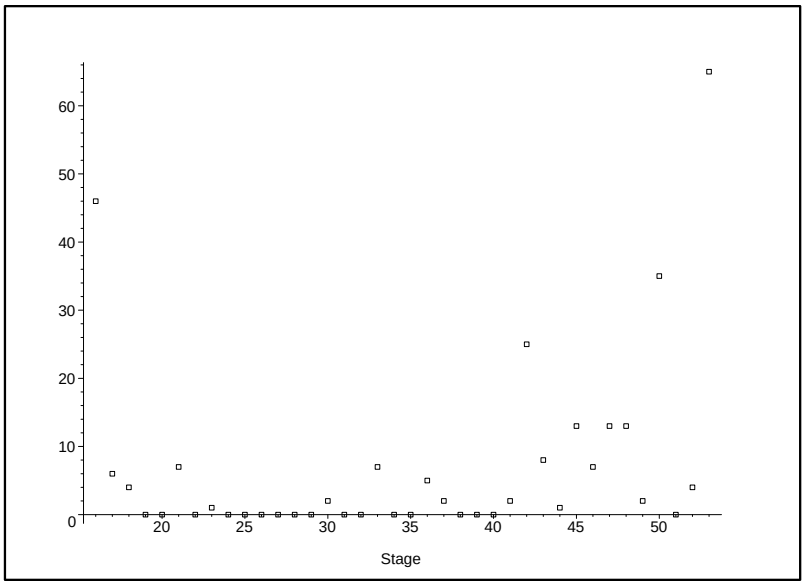

Fig. 2. Difference in number of successfully typed object per stage (post stage 16)

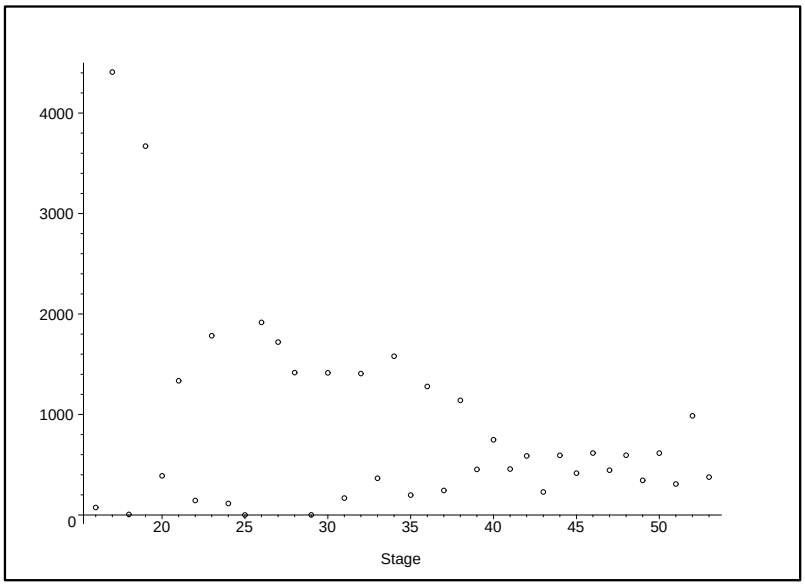

Fig. 3. Number of changed Failure messages per stage (post stage 16)

and a "low" curve corresponding to the more idiosyncratic bottom-up passes; this distinction is apparently at least until stage 40, when the two curves get very close to each other. The one outlier in the bottom-up process is stage 27, as the relational operators $=$, $<>$, $<$ and $\leq$ together caused quite a few failures. We believe that this graph lends credence to our methodology of combining top-down and bottom-up approaches.

In Table 2, we show the most common Failure mode for our inferencer after a successful bottom-up stage. This is the deciding factor for what will be implemented in the next, top-down, stage. First, it is clear that these numbers are generally decreasing, which means that the bottlenecks "spread out" over more cases. The large drop after stage 12 reflects that Maple procedures are not uniform, and

that while assignments are most common, they account for just over half. The drop at stage 20 actually reflects two things: that `if` statements are quite common, and that in stage 17 builtin procedures were split out into cases. That sometimes the numbers go back up reflects that handling more cases sometimes causes a refocusing of the reasons for failures into a few common bottlenecks. By stage 53, while there are still 12050 objects to "type", the most common source of failure accounts for just 4.5% of the total. The top 10 however do account for 32% of the failures, so there still are quite a lot of commonalities.

We will make our complete results available as a technical report, but this already spans roughly 150 pages of "data".

5 Discussion

Our systematic approach had the side-effect of generating a considerable amount of data. Some of it is pure trivia, but quite a lot was rather surprising to us. In this section, we provide a sampling of what we have found, first providing some explicit examples, and then concentrating on those items which were most unexpected.

5.1 Examples

5.1.1 `Im/abs`

A trivial example is that for the procedure `Im/abs`, which is a special case of the `Im` function (the imaginary part) applied to a symbolic absolute value. Since the result of an absolute value is always a positive real, `Im/abs` is a trivial procedure which simply returns 0 always.

Unsurprisingly, the analysis of this procedure is not difficult. What is perhaps more surprising is that, as a consequence of our design choices, the return value is actually hardcoded in the contract: `Function([],Static(0))` expresses succinctly the fact that this is equivalent to the constant zero function.

5.1.2 `codegen/joinprocs/isAssign`

A short utility procedure used by the `codegen package`, this routine is analyzed to be of "type" `Function([OpAble(0)],truefalse)`. That is, this function accepts a single argument (say, `x`), and the function requires that the Maple command `op(0, x)` be defined. The symbol `truefalse` indicates that the result is a boolean value.

The `OpAble(0)` predicate, which indicates that Maple's `op` command may be invoked to extract the 0th argument, is really the equivalent of a type class in Haskell.

5.1.3 `codegen/C/function/argtype/irem`

This example exhibits need for first-class types in our contract language. The contract generated is

```
Function([Parameter(1,"mode")],Static(Symbol("integer")))
```

This specifies that the object in fact a function from an arbitrary input to a *type* encoded as a plain value.

stage	occurrences	Failure mode
2	11023	cannot handle PROC
4	11023	cannot handle GLOBALSEQ
6	8955	cannot handle nonempty LOCALSEQ
8	9419	cannot handle nonempty OPTIONSEQ
10	9708	cannot handle nonempty PARAMSEQ
12	9544	cannot handle STATSEQ
14	4502	cannot handle ASSIGN
16	4414	cannot handle ASSIGNEDNAME
18	3671	cannot handle IF
20	1653	InferType not implemented
22	1531	cannot handle PARAM
25	1917	cannot unify non-Static types
27	1417	VerifyBoolean not implemented
29	1416	cannot handle LOCAL
31	1407	no type for builtin type
33	1419	cannot re-assemble 2-arg ASSIGN
35	1120	cannot re-assemble 2-arg FUNCTION
37	1141	cannot handle TABLEREF
39	748	no type for builtin op
41	616	unknown function call OpType
43	594	no type for builtin nops
45	628	cannot handle ERROR
47	608	cannot handle ARGS
49	669	cannot re-assemble IF
51	788	cannot handle RETURN
53	549	cannot handle FORFROM

Table 2
Most frequent failure after a bottom-up stage, leading to a top-down stage

5.2 Assertions

Hinze, Jeuring, and Löh are very specific in stating that their use of the term "contract" implies that a contract affects program behaviour.[3] We do not share this view of contracts, and here we concentrate on what they refer to as *static properties*. However, as we have to deal with many more issues, these properties fall far outside the usual realm of properties considered for type systems. Nevertheless, it would be an easy task to implement an "assert" command similar to theirs which, given a procedure and an (inferred) contract, returns a procedure which checks that the contract is satisfied before and after execution. Such a construct may lead to more information-rich error messages, and is worth further investigation.

5.3 Artifacts

Any piece of software that has been developed continuously for a quarter-century will necessarily have accumulated baggage along the way. Our investigations have unearthed a host of oddities, few of which are outright bugs, but many of them interesting artifacts nevertheless.

The Maple system has evolved greatly over the years. Quite a few older features have been deprecated, and mention of their existence is all but banished from much of the present documentation. However, because of the sheer size of the library and the need for backwards compatibility, it cannot all be "upgraded" to employ only newer features.

Names defined in a Maple archive are stored in records called *dot-m* files, whose load is triggered when the associated name is first seen. Though modern Maple code is designed in such a way that there is at most one global name defined per dot-m file, this was not always the case. It is therefore possible for certain global names in Maple to be defined from the main Maple repository only if some other name, whose relationship to the first may not be obvious, is read first. Among the names defined in this "hidden" way are the globals `a` and `b`; the concern this fact may stir should be dampened somewhat by the fact that they are buried under the name `tttesting1/__a`, which the casual user is unlikely to use. Also, names used to not be automatically defined, and so one had to use the routine `readlib` to achieve this; as this practice was discouraged quite some time ago, we were quite surprised to discover 80 calls to `readlib` remaining in the library.

One may also wonder, why are there so many integer constants stored in the library? Some are so-called *magic constants* which control the behaviour of certain routines (provide thresholds, etc). Others are pure artifacts: when the tables for integral transforms are saved, the number of entries in each table is also saved in a separate name, yet this name is not referenced anywhere else in the library! So those integers serve no effective purpose.

5.4 Modules

In modern Maple (i.e. starting with Maple 6), modules are one of the most important structuring mechanisms for larger pieces of code, including the Maple library.

Though in following our greedy approach we have so far ignored top-level modules, given the special focus on modules in the last several Maple releases we might still expect to encounter many references to modules in the analysis of other top-level names.

As such, it is quite surprising that through 53 passes in the library, modules have not been at top of the list of bottlenecks. It would appear that "old" Maple (i.e. the code saved in top-level names not assigned to modules) does not call "new" Maple very much. Perhaps this indicates that newer routines are somehow less "central" to the system, which would be reasonable for a mature system. Or perhaps this indicates that the refactoring and maintenance of older routines is not an active concern of Maplesoft.

5.5 Code Templates

There is a long tradition of run-time code instantiation in Maple. The modern method is to use lexically scoped procedures (or modules), but this does have a certain overhead for the generated procedure. The older method, of using `subs` directly into a template `proc` is sometimes still used for this purpose. However, this older method fell into disfavour because it was accident-prone: since it used explicit names for the substitution, it was possible to be unlucky and have accidental name captures. Efforts to avoid these accidental name captures led to some rather creative names used as placeholders in some procedure templates.

There is, however, a modern twist: since a Maple string may never have an assigned value, one may safely use a string in place of a name for the substitution parameter and avoid accidental capture. However, since the string is usually likely to be a different type than its substituted value, this has the side-effect of having the code template being *invalid* code, i.e. code which cannot be run on its own without generating a nonsensical error, but which works fine after the substitution is performed. `'dsolve/numeric/MB/solnproc'` is one such a routine, and they cause many headaches for our inference engine!

5.6 Monoids and other algebraic structures

One might hastily conclude Maple's library is entirely free of algebraic structures, and this would be quite inaccurate. Nevertheless algebraic structures do seem to be present implicitly rather than explicitly. They may appear in two ways: via a natural morphism, or via correctness conditions. In the first case, we find a natural refinement morphism between the specification of an algebraic structure and a computational implementation. It seems quite difficult to infer such algebraic structures, as frequently such morphisms are not invertible. In the second case, there is also a natural refinement morphism present, but also natural correctness conditions — for example division can only be used when it exists and is well-defined. In this case, the algebraic structure seems to be an emergent property of the implementation.

6 Conclusion

Among the more surprising results of our investigation is the high importance in the system of traditional "computer science" data structures such as lists, sets, functions, variables, and conditionals. This is in contrast with the rarity of basic algebraic structures such as sums, products, or more complex structures such as monoids and groups. But just as interesting is the pervasiveness of symbols, and of functions with variable numbers of arguments. While we also found a lot of dynamic type checks, it is still unclear to us whether these are dynamic by necessity, or simply for convenience to the programmer.

On the other hand, we have found substantial use of first-class types (i.e. types as values) and dependent types; there is also considerable use of *ad hoc* polymorphism (through `op`, `nops`, etc). It is not clear to us whether these are fundamental or whether a solid implementation of pattern-matching, with proper syntactic support (unlike what is offered by `typematch`) would remove the need for these.

The frequency of assignments involving single-use local names suggests that Maple sorely needs a `let` construct. Surprisingly, the data also suggest that Maple is arguably somewhat more functional than imperative, as there are more uses of `map` than of `for` loops.

Finally, it seems to us that many of the currently unhandled built-in functions may depend upon some sort of generalized "Expression" type. Such a dependency would be somewhat disheartening, as the generality of such a type would impede wide-scale inference of useful static contracts involving these builtin functions.

References

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

[2] Goguen, J., *Types as theories*, in: G. M. Reed, A. W. Roscoe and R. F. Wachter, editors, *Topology and Category Theory in Computer Science*, Oxford, 1991 pp. 357–390.

[3] Hinze, R., J. Jeuring and A. Löh, *Typed contracts for functional programming*, in: *Proceedings of FLOPS*, 2006.

[4] Meyer, B., *Applying "design by contract"*, Computer **25** (1992), pp. 40–51.

[5] Meyer, B., "Eiffel: The Language," Prentice Hall, 1992.

[6] Nielson, F., H. R. Nielson and C. Hankin, "Principles of Program Analysis," Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[7] Peyton Jones, S., "Haskell 98 Language and Libraries," Cambridge University Press, 2003.

[8] Pierce, B. C., "Types and programming languages," MIT Press, Cambridge, MA, USA, 2002.

[9] Pierce, B. C., editor, "Advanced Topics in Types and Programming Languages," MIT Press, 2005.

[10] Taha, W., "Multi-Stage Programming: its theory and applications," Ph.D. thesis, Oregon Graduate Institute of Science and Technology (1999).

[11] Watt, S. M., *Aldor*, in: J. Grabmeier, E. Kaltofen and V. Weispfennig, editors, *Computer Algebra Handbook: Foundations, Applications, Systems*, Springer Verlag, 2003 .

A Maple Glossary

We provide a brief description of a few of the commands or paradigms specific to the Maple programming language that have been referenced in the text.

- `op`: Returns the operands of any structured expression (e.g. the elements of a list or set, or the arguments of a function call.)
- `nops`: Similar to `op`, but returns only the *number* of operands.
- *inert form*: A method of representing Maple code as a data object consisting of unevaluated (inert) function calls, equivalent to an abstract syntax tree. This may be programmatically manipulated with no fear of accidental evaluation.
- `ToInert, FromInert`: A means of converting live code (e.g. an expression, procedure, or module) into an **inert form** and vice-versa.

Table A.1 provides a short synopsis of the data structures associated with the more esoteric names among the inert form names used throughout the text.

name(s)	data structure
ASSIGNEDNAME	Name with an assigned value, as opposed to an (unassigned) symbol
CONDPAIR	"Conditional pair": a boolean condition and associated statement, found only inside an IF structure
DCOLON	Procedure parameter or local variable with an explicit type annotation
EXPSEQ	"Expression sequence": a self-flattening sequence of expressions
FORFROM	A for/while loop: both for loops and while loops are special cases
INTPOS, INTNEG	Positive and negative integers, respectively
TABLE	Hash table
TABLEREF	Index into a hash table
UNEVAL	Wrapper around an expression which delays evaluation

Table A.1
Data structures corresponding to inert form tags