Runtime code generation to speed up spreadsheet computations

Thomas S. Iversen, zensonic@diku.dk

July 30, 2006

Figure 1: Taming the compiler dragon inside spreadsheet engines. The picture is taken from [2] and edited in The Gimp.

Abstract

In this thesis, a small spreadsheet calculation engine written in C#, is augmented with runtime code generation (RTCG), I/O methods, a GUI and scripts. The resulting spreadsheet system is named TinyCalc. The runtime code generator compile formula expressions to IL assembler in .NET code while inlining subexpressions and deducing types of values to avoid creating typechecks in the resulting code. Moreover, it tries to avoid creating temporary objects for intermediate values and, finally, it tries to let the basic objects of calculation in .NET, `double` and `string` stay directly on the runtime evaluation stack between operations. The speedup of doing so is measured using simple benchmarks and it is concluded that TinyCalc, while simple, is able to outperform Gnumeric and Open Office Calc *without* using RTCG. *With* RTCG, TinyCalc approaches and sometimes perform on par with Excel, when not counting the overhead of doing RTCG. It is also concluded that TinyCalc needs to rethink the way it does Matrix operations. The thesis ends with a section on the perspectives for TinyCalc and RTCG.

Contents

List of Figures

Notational conventions

Throughout the thesis the notational conventions listed in table 1 is used.

Notation	Description
Class	Notation and typesetting used for classes.
`Method`	Notation and typesetting used for methods.
`Cell`	Notation and typesetting used for cells and cell references.
`Filename`	Notation and typesetting used for filenames and paths.
Abbreviation (abbr)	Denotes that "Abbreviation" are abbreviated as "abb".
`application`	Denotes an external application or utility, ie. `ildasm`.

Table 1: Notational conventions

Furthermore the following notation denotes cell definitions:

```
[<sheetname> '!'] (<cellreference> | <cellarea>) '=' <expression>
```

where

```
sheetname := "Sheet" integer
cellreference := A1-style reference | R1C1-style reference
```

Section 4.1.2 gives an overview of the A1 and R1C1 reference styles. `cellarea` and `expression` are defined in terms of the BNF grammar used by TinyCalc to define formulas, see section A.3. Two examples of the notation can be seen below.

Example 1 Example of a cell definition

```
A1 = SUM(B1:B5)
```

This denotes that the *single* cell `A1` contains a formula `=SUM(B1:B5)`.

Example 2 Example of a cell definition

```
Sheet2!A1:B2 = MATINV(Sheet1!A5:B6)
```

This denotes that the cell*area* `A1:B2`, that is the cells `A1`, `A2`, `B1` and `B2`, in `Sheet2` *all* are defined in terms of the single formula `=MATINV(Sheet1!A5:B6)` which computes the inverse of the matrix defined in `Sheet1!A5:B6`.

1 Foreword

This thesis was written by Thomas S. Iversen (`zensonic@diku.dk`) in the period December 2005 - July 2006 at the Department of Computer Science (DIKU), Universitetsparken 1, DK-2100 Copenhagen Ø to obtain a degree as master of science.

A small spreadsheet system named TinyCalc is augmented with runtime code generators and this implementation is used to investigate code generation in spreadsheets. TinyCalc is benchmarked against its own basis implementation which uses evaluation and against then professional spreadsheet systems, Excel, OpenOffice Calc and Gnumeric.

Readers of this thesis should be familiar with spreadsheets in general; know what they are and how they are used. Furthermore, a good understanding of computer science in general is required. Lastly some knowledge of C# and IL assembly and assembly code in general is also required, preferably in conjunction with Visual Studio. Suggestions for getting up to speed on C#, IL and Visual Studio is [34], [20], [22] and [21]; books which have been the basis for the author.

This thesis also exist in a PDF version with click able hyper references and can be found on the accompanying CD-ROM (see section A.11). The source code produced has been omitted from the paper version of the thesis, but a ready made PDF file with all the source is located on the CD-ROM. Besides that, the thesis, source code and tests can be located at

`http://www.dina.kvl.dk/~thomassi/thesis/`

The author of this thesis would like to thank his supervisors, Peter Sestoft, ITU (`sestoft@dina.kvl.dk`) and Torben Æ. Mogensen, DIKU (`torbenm@diku.dk`) for their good ideas, the valuable criticism and their input to this thesis. Also thanks to Lars Josephsen, KVL, who provided the Excel license for this thesis. Finally, thanks to my wife Maibritt and my kids Mathias and Benjamin for supporting me during this period.

2 Background

One of the most appealing features of spreadsheets is that they enable one to *easily* perform similar calculations on data with identical properties. An example of this could be a spreadsheet with one row per person, the person's name in column 1, his hourly salary in column 2, the number of hours he has worked this month in column 3 and finally a taxdeduction formula in column 4 which uses value in column 2 and 3 to calculate the monthly wage for the person in this row. Having done this for one person, one can simply *copy and paste* the formula for all the other persons. Spreadsheets are excellent at representing repetitive data dependencies. Almost all spreadsheet systems in existence today provide extensive editing capabilities in this regard.

Being good at representing repetitive, rigid structures, spreadsheets are widely used. Spreadsheets are however not suited for demanding computations. In these cases a custom developed application often provides faster execution times and/or smaller memory usage. While most people, without a degree in computer science, can use and understand spreadsheets, not many can program an application when the problem becomes too large and complex for the spreadsheet system to handle.

When Peter Sestoft discovered that people in the insurance business sometimes struggled with long recalculation time of spreadsheets, he formulated the idea of runtime code generation (RTCG) in spreadsheet systems. Section 5.1 gives an overview of runtime code generation.

This thesis investigates whether or not, RTCG can speed up spreadsheets in the situation where the spreadsheet is (becoming) too complex for the spreadsheet calculation engine to recalculate in reasonable time.

3 Thesis, goals and priorities

The main hypothesis of this thesis is that the time it takes to recalculate spreadsheets can be reduced by using runtime code generation in the spreadsheet application. Most notably for spreadsheets containing many identical formulas. Two formulas are considered identical if they can share the same internal formula representation. Runtime code generation in spreadsheets can be explored in the following two ways:

- Use of an existing, full blown system as a vehicle for exploration, or
- An experimental spreadsheet system which could be built from the ground.

Using an existing system like Excel or Open Office Calc requires that the flow of formula evaluation in these systems is intercepted and augmented with some kind of mechanism which uses RTCG for evaluation. Doing so, would enable one to concentrate on RTCG and eliminate the need for spending time on internal data representation, the GUI required to edit values interactively and the I/O routines needed to get data in and out of the system. The downside to doing things this way are, however, not negligible. It might not be possible to intercept the formula evaluation mechanism or it might require a great deal of guesswork on the inner workings of the spreadsheet system.

While the above approach certainly is worth considering the background for this thesis has been to build a small, customized spreadsheet system geared towards investigation of various hypotheses. This will give a more thorough and faster understanding of the various problems as well as total control over the system. The disadvantage is that the application has to be built from the ground.

Building a spreadsheet system that provides data structures and methods for recalculation of data, a GUI for manipulating data and I/O methods to get data in and out of the system is in itself a non-trivial piece of work. To remedy this situation Peter Sestoft has provided the infrastructure for a small spreadsheet system called CoreCalc. CoreCalc has support for text strings, floating point values and matrix values. This code will provide the basis for development. As Peter Sestoft had RTCG in mind when developing the code, the formula representation in CoreCalc is already geared towards RTCG. This will be discussed further in section 4.2.

The goals for the thesis are:

1. Implement a spreadsheet system prepared for RTCG. Specifically, it should support the concept of identical formulas. This system will be called TinyCalc.[1]
2. Develop a GUI to the system geared towards debugging, investigation and development of RTCG in spreadsheets.
3. Implement input and output support for a widely used format of spreadsheets. More explicit the formats used in Gnumeric, Open Office Calc (OOCalc) and Excel will be taken under consideration.
4. Consider how to maximize sharing of formulas when inserting and deleting rows and columns as well as moving data around.
5. Extend TinyCalc with RTCG and consider RTCG for formula expressions as well as for the functions involved in these expressions.

[1] The first commercial spreadsheet system was VisiCalc in 1987. The name TinyCalc is a kudos to VisiCalc's inventors while still describing the purpose of the application.

6. Test the implementation of RTCG in TinyCalc
7. Evaluate gains in recalculation times in various situations and accept or reject the hypothesis for each situation.
8. Compare the recalculation times to those obtainable in professional spreadsheet systems, most notably Microsoft Excel 2003, Open Office Calc 2.0 and an Open Source system named Gnumeric.
9. Formulate when and when not to use RTCG in spreadsheets.
10. Conclude on the entire process.

The list above is *not* prioritized. It does, however, more or less depicts the natural workflow. That said, the following applies with regards to the priorities. (1) Implementation of a small system is already partly done by Peter Sestoft. The top priority is getting this finished. This includes extending the existing system with new functionality as well as considering if the code Peter Sestoft has written needs to be adjusted for this thesis. (6) Correctness of the calculations is important, but a full thorough test will not be conducted. Instead, regression tests will be developed during development. (4) While it is important to consider the implications of performing various operations on a spreadsheet with regard to their data representation, their actual implementation is of lower importance in this system. (3) It is important to be able to get actual spreadsheets and workbooks in and out of the system, but the performance of the I/O routines is not important. (5) Finally a high priority is to analyze and implement RTCG for the formula expressions themselves, as this alone will give information about the complexity and the possible speed gains of RTCG.

4 Building a spreadsheet system

4.1 Anatomy of spreadsheets

To understand what spreadsheet systems are by today's standard, three modern spreadsheet applications are inspected. These are Excel 2003 (version 11.5612.5606) by Microsoft Corporation, Open Office Calc version 2.0 by Sun Microsystems Inc. and finally Gnumeric 1.5.90 developed by the GNOME Foundation. These three systems are hereafter named Excel, OOCalc and Gnumeric.

A spreadsheet consists of one or more *sheets* which is combined into a unit. In some systems, this combined collection of sheets is called the *workbook*, in others, it is merely called a spreadsheet. In this report we will adopt the terminology workbook. Each sheet in the workbook consists of *cells*, arranged in a grid of *rows* and *columns*.

A cell is the basic unit for simple calculations and can hold a *value*. A *cell area* is the basic unit for matrix calculations. Cell areas define rectangular collections of cells. A cell can either hold a *constant* value or a *formula* which, when evaluated, gives the cell its value. Most spreadsheet systems implement *string* constants, *integer* constants, *floating point* constants and *boolean* constants. Most spreadsheet systems also support more advanced values like *matrix* values, *complex number* values and *date/time* values. Lastly most spreadsheets has support for *named cells* and *named ranges* where a user defined name is assigned to cells, cell ranges and cell areas and this name can then be used in formulas. This provides a convenient abstraction level when "building" a spreadsheet and removes need for hard coded cells in formulas.

To distinguish constant string values from formulas, most spreadsheet systems have adopted the convention that cell formulas start with the character '=' followed by an *expression* telling the spreadsheet how to calculate the value. There is a difference between cell `A1` containing the constant value `5` and it containing the formula `=5` that, when evaluated, evaluates to `5`. Expressions consists of *operators* and *operands*. Examples of operators are addition, subtraction, multiplication and division. Operands can be either constant values or references to other cells, giving the possibility to create dependencies among cell values. Early spreadsheet systems allowed only one sheet, so only references to cells in that sheet were possible. Modern spreadsheets can have many sheets and so cell references are allowed to refer to any sheet in the workbook. This allows very sophisticated dependencies to be created. Some systems even allow references to other workbooks, loading these on demand.

Besides the common operators such as addition, subtraction, and so forth, spreadsheet systems contain built-in functions. Normally, these include, but are not limited to, mathematical, statistical and financial functions. Most spreadsheet applications also have some kind of report and graph generation facility built in.

Many modern spreadsheet applications also support some kind of computer language (Basic, Pascal or a scripting language). This language is used to define *functions* or *subroutines* that can be called from formula expressions. A user defined function (UDF) for calculating the mean value of an array of single precision numbers in Visual Basic (VBA) could look like:

Example 3 Example of a user definable spreadsheet function

```
Function Mean(Arr() As Single)
   Dim Sum As Single
   Dim i As Integer
   Sum = 0
   For i = 1 To UBound(Arr)
      Sum = Sum + Arr(i)
   Next i
   Mean = Sum / UBound(Arr)
End Function
```

Most spreadsheets including the three under investigation use some variant of an imperative language but object oriented and functional variants do exists.

Calculations in a workbook are normally performed either automatically and instantaneously when a value of a cell is changed or explicitly when requested by the user. For performance reasons both Excel, Gnumeric and OOCalc tries to recalculate only cells *affected* by the cell update contrary to recalculating *all* cells. This way of recalculating the workbook is a challenge to implement efficiently and correctly. The former because the dependency information with even very few formulas can be prohibitory large. The latter because "problematic" constructs like `A1=IF(A2;A1;A3)` is allowed. This formula *might* introduce a cyclic dependency in the workbook. User defined functions might also cause problems as they might not be *required* to include cell referenced in the body of the UDF in the parameter list to the UDF. This is what causes all kinds of weird behavior in Excel, see section 8.

4.1.1 Cell references

References in an cell formula can be either *absolute* or *relative*. An example will show the difference. Assume that cell `A3` is expressed as:

$$\texttt{A3 = A1 + A2} \tag{1}$$

This is a formula with two relative cell references. It is possible to express the same formula using absolute cell references. For instance

$$\texttt{A3 = A\$1 + A\$2} \tag{2}$$

Even though these two formula calculates the same expression as they are shown here, they are quite different conceptually. When copied to cell `A4` formula (1) becomes

$$\texttt{A4 = A2 + A3} \tag{3}$$

whereas the (2) remains

$$\texttt{A4 = A\$1 + A\$2} \tag{4}$$

Depending on the purpose of the copy, either representation might be desirable. The '`$`' fixates the row index when copying or moving the formula to a new location. By duality the same semantic works for columns.

Investigation of OOCalc, Microsoft Excel and Gnumeric shows that all three applications support both relative and absolute addressing. All three spreadsheets also support inter-sheet references. Excel and Gnumeric use the ! sign as marker:

$$\texttt{Sheet1.A1 = Sheet2!A1} \tag{5}$$

whereas OOCalc uses a dot (.) to denote the same. While Excel and Gnumeric only support absolute sheet references OOCalc support both absolute and relative sheet references, with relative sheet references being the default. One explicitly has to request absolute sheet references as in

$$\texttt{Sheet1.A1 = \$Sheet2.A1} \tag{6}$$

which, when copied to Sheet3.A1, then yields

$$\texttt{Sheet3.A1 = \$Sheet2.A1} \tag{7}$$

4.1.2 Cell reference styles, A1 vs R1C1

Cell references as introduced in section 4.1.1 are called A1 style references. All three systems understand this style of references. An alternative to this style is the R1C1 style in which formula (1) becomes[2]:

$$\texttt{A3 = R[-2]C + R[-1]C} \tag{8}$$

R1C1 can express both relative and absolute references. Absolute references assume origo at (1,1) in the upper left corner. The square brackets denote relative references and negative numbers denotes columns to the “left” or rows “above”. The absence of a number for either a row or column index means that the indexing is relative and denote the current row or column. So expressing formula (4) in R1C1 style we obtain:

$$\texttt{A4 = R1C + R2C} \tag{9}$$

When copying formula (8) to cell A5 it remains:

$$\texttt{A4 = R[-2]C + R[-1]C} \tag{10}$$

It should be noted how formulas (8) and (10) are identical whereas (1) and (3) are not. While A1 style references might be easy to understand, they have the drawback that their indexes need to be updated when a formula is copied. References in R1C1 style does not have this problem. This is caused by the fact that in a R1C1 type reference the index denotes two different things depending on whether the reference is absolute or relative.

It is easily seen that R1C1 and A1 are equivalent in the sense that an arbitrary A1 type reference can be converted to an R1C1 type reference, and vice versa.

Only Excel, supports displaying an entry of R1C1 references. They are not selected by default, but Excel can be told to use them. This, however, disable the support for the A1 reference style. Investigation shows that Excel uses R1C1 style references internally, at least when saving spreadsheets, no matter what the GUI accepts when editing. Both Gnumeric and OOCalc use A1 style references for saving formulas.

4.1.3 Updating references

When a cell is moved to a new location, all references to this cell have to be updated automatically. The word “moved” is used in a very broad sense here. When rows

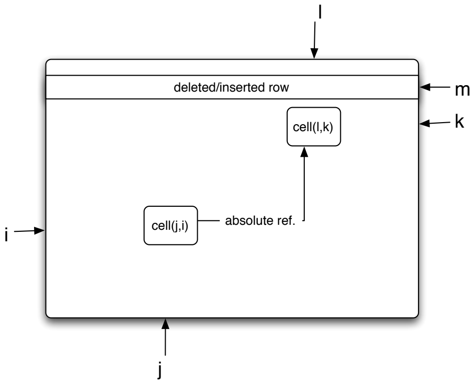
[2] It should be noted that R[-2]C is a valid shorthand notation for R[-2]C[0]. In fact Excel always shortens references according to this shorthand notation. Similarly R[0]C[0] = RC

and columns are inserted or deleted, some cells are moved. When an entire sheet is inserted, renamed or deleted a whole sheet of cells are “moved”.

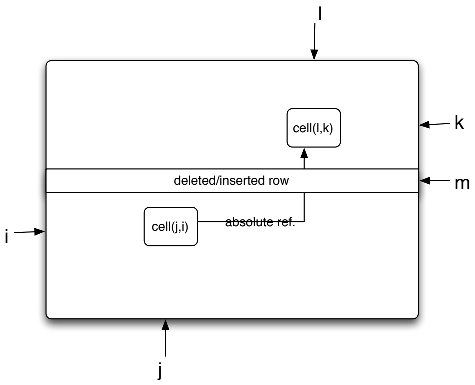
It is possible to categorize moves according to what has to be done to a cell reference because of the move.

Assume that cell $C[i, j]$ in row i and column j contains an reference to a cell $C[k, l]$ in row k and column l. Assume furthermore that $i \neq k$ and consider what happens when deleting or inserting a row. When deleting or inserting a new row m, some or all cells get new row names. More precisely, four cases can be formulated, two for absolute reference and two for relative references. For absolute references:

$m < k$: When deleting or inserting row m, where $m < k$, cell $C[k, l]$ become either $C[k - 1, l]$ or $C[k + 1, l]$ respectively and as a consequence cell $C[i, j]$ needs to be updated with a new reference:

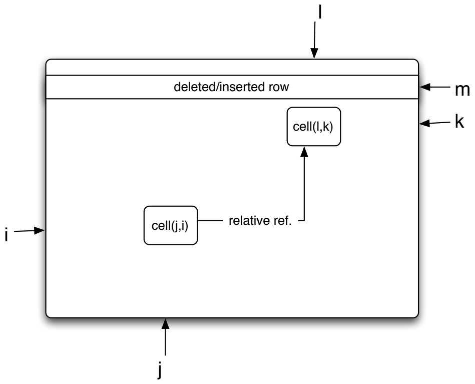


$m > k$**:** When deleting or inserting row m, where $m > k$, cell $C[k, l]$ continues to be cell $C[k, l]$ and therefore cell $C[i, j]$ needs not be updated with a new reference:

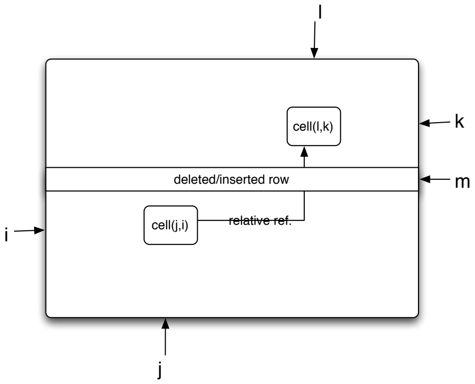


For relative references:

$m \leq \min(k, i)$ *or* $m > \max(k, i)$: When deleting or inserting row m, where $m \leq \min(k, i)$ *or* $m > \max(k, i)$, cell $C[k, l]$ become either $C[k-1, l]$ or $C[k+1, l]$ respectively *and* cell $C[i, j]$ becomes either $C[i-1, l]$ or $C[i+1, l]$ respectively and as a consequence of relative references cell $C[i, j]$ needs not be updated with a new reference:



$\min(k, i) < m \leq \max(k, i)$: When deleting or inserting row m, where $\min(k, i) < m \leq \max(k, i)$, cell $C[k, l]$ continues to be cell $C[k, l]$ but cell $C[i, j]$ becomes either $C[i-1, l]$ or $C[i+1, l]$ respectively and as a consequence of relative references cell $C[i, j]$ needs to be updated with a new reference.



The same situation goes for column based insertions and deletions. If one think of the sheets in a workbook as stacked, forming a third dimension, a situation similar to the above presents itself when deleting or inserting sheets in a workbook. It is not hard to list all cases, but it will be tedious to write and boring to read so a full analysis of all cases is not presented.

4.2 Implementing a spreadsheet system

Having identified the main components of modern spreadsheet systems, it is now possible to develop a little spreadsheet system for this thesis. It will be augmented with RTCG in a later chapter, but decisions and directions taken in this chapter has to consider this fact, so problems later will not arise because of this.

4.2.1 Features

The spreadsheet system is going to be named TinyCalc and is going to have the following features.

- The concept of workbooks and multiple sheets.
- Intersheet references as described in section 4.1.1.
- Support for floating point numbers (double, 64-bit IEEE754[16]) and arithmetic on these numbers. Integers are supported by casting them to double numbers and vice versa.
- Support for matrices of real numbers and functions to perform calculation on matrices.
- Support for text strings and operators to perform string concatenation.
- A basic but usable GUI.
- Support for both A1 and R1C1 style references.
- A basic grammar for formulas.
- A command line interface enabling one to perform calculations, tests and benchmarks from the command line. This is needed for automatic test and benchmarks.
- The possibility of loading and saving workbooks.
- A list of functions which can be used when constructing formulas. The list should be extensible and easy to maintain.
- Support for recalculation and detection of cyclic dependencies.
- Support for type checking formulas during recalculation.
- The choice of performing recalculation when updating a cell or manually by a keystroke.

Note that this feature list indicates support for pretty printing and formatting of data have not been considered important.

4.2.2 Equivalence of formulas

As noted earlier, sharing of formula expressions are crucial for RTCG in spreadsheets. Only equivalent formulas can be shared. While these calculations

$$(A1 - B1) + C1 = A1 - (B1 + C1) = -B1 + (C1 + A1)$$

hold as algebraic identities they do not necessarily hold when calculating with floating point numbers, which have limited precision. So equivalence of formulas is not algebraic equivalence. Even when it is possible to rearrange formulas so that they still produce the exact same result using floating point arithmetic, it is not a good idea to do so. Firstly, it takes a considerable amount of time to determine if a formula can be rearranged into a new formula equivalent to one already seen. Secondly, the user might be confused when entering `A1+B1+C1` and it gets rearranged to `B1+A1+C1`.

So two formulas F and G in TinyCalc are equivalent when their abstract syntax tree are identical. Equivalence testing two abstract syntax trees is nontrivial. To circumvent this it is assumed that if the two strings obtained by expressing the abstract syntax trees as formulas using R1C1 style references are identical, so are their abstract syntax trees.

Using this definition, formulas copied and pasted in a spreadsheet will be considered equivalent. This should be more than sufficient to exploit the amount of sharing of formulas possible in a spreadsheet as most identical formulas indeed are the result of copy and paste operations.

4.2.3 Representing spreadsheets

While the cells in a spreadsheet conceptually and visually are arranged in rows and columns, their internal representation might not be. There are in fact two common ways of constructing the internal representation of a spreadsheet:

1. Using a two-dimensional array or
2. Using one or more Direct Acyclic Graphs (DAGs) where the vertices represent cells and the edges represent dependencies between cells.

Each has its strength and its weaknesses. The array representation provides for O(1) lookup of a cell given its row and column index. This is not possible with a DAG. A DAG, however, expresses dependencies in a way that recalculation of the entire sheet after update of a single cell can be done easily by only recalculating the cells which actually need recalculation. This is not as easy with an array structure. The biggest disadvantage of DAGs however, is that they do not work well with constructs that change dynamically during runtime. Consider

$$\text{A2} = \text{IF(A3+A4=5;A6;A2)}$$

The above is a perfectly valid expression in Excel, OOCalc and Gnumeric even though it *might* produce a cyclic dependency during recalculation. The expression can not be represented using a DAG as cycles are not allowed in a DAG and a DAG has to represent both branches of the `IF` statement. A cycle will turn the DAG into a general Graph which would make it impossible to use topological sorting to generate the sorted list used for proper recalculation order. It is known that Excel and Gnumeric uses dependency information to recalculate only cells affected since last recalculation. How this is done if a DAG is used and how this DAG would represent the `IF` construct above is unknown.

It is possible to augment the array structure with dependency information and it is also possible to maintain a DAG besides the basic array structure in an effort to obtain the best of both representations.

4.2.4 Recalculation times and strategies

Consider a spreadsheet with n cells of which m cells are used and consider a change in a cell s which affects k cells. To perform a recalculation in a DAG this algorithm is used:

```
RECALCULATE-DAG($G, s$)
while $Q \neq \emptyset$
  $Q \leftarrow \{s\}$
do $u \leftarrow head[Q]$
    REEVALUATE($G, u$)
    DEQUEUE($Q$)
    for each $v \in Adj[u]$ in topological sorted order
    do
        ENQUEUE($Q, v$)
```

It should be noted that the above algorithm skips the finer details in obtaining the next cell in topological sorted order. It is crucial that the cells are recalculated in dependency order. This is not an easy task, but can be done efficiently using a topological sorting of G. Sorting the graph G using a topological sort will give the algorithm a running time of $\Theta(V+E)+O(k)$, where V is the number of vertices in the G and E is the number of edges in G. Sorting G from scratch with each update to G is known as offline topological sorting. It is also possible to do online topological sorting of G where the previous sort of G is used as basis for sorting G after the update. Two articles [9] and [32] present algorithms for doing online topological sorting. The first give an $O(V^{2.75})$ algorithm, the other a $O(\delta \log \delta)$ where δ are the number of nodes needing recalculation. So the recalculation time using a DAG is $O(k)$ but requires a non-trivial amount of time to maintain the dependency invariant when a cell is updated, deleted or inserted.

The array structure does not give the luxury of knowing the dependencies in a backwards manner, that is “which cells depend on this cell?”. An array structure instead provides information on which cells this cell depends, ie dependencies flowing forwards. To recalculate a spreadsheet when it is represented as an array, a recursive depth-first approach is used. To avoid recalculating the same (sub)expression over and over again a status field and a cached value for each cell is maintained. The algorithm can be seen below. Figure 2 depicts the simple state machine of the algorithm.

```
RECALC-ARRAY($A$)
for $i = 1 \ldots n$
do
    $state[i] \leftarrow dirty$

for $i = 1 \ldots n$
do
    if $A[i].e \neq$ NIL
       then $A[i].v \leftarrow$ RECAL-CELL($A, i$)
```

```
RECALC-CELL($A, i$)
if $state[i] == dirty$
  then
       $state[i] \leftarrow inprogress$
       $v \leftarrow$ RECALC-CELLVALUE($A, i$)
       $state[i] \leftarrow uptodate$

return $v$
```

```
RECALC-CELLVALUE($A, i$)
if typeof ($A[i].e$) == $ScalarExpression$
  then
       return $A[i].e$
  else
       // for this example assume simple
       // addition of two cells.
       $cellref1 \leftarrow A[i].e[1]$
       $cellref2 \leftarrow A[i].e[2]$
       $v1 \leftarrow$ RECALC-CELL($A, cellref1.index$)))
       $v2 \leftarrow$ RECALC-CELL($A, cellref2.index$)))
       $v \leftarrow v1 + v2$
       return v
```

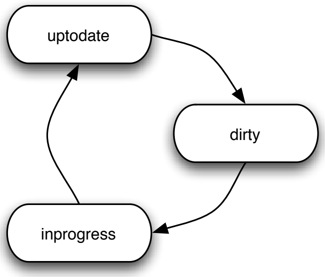


Figure 2: State machine implemented by Recalculate-Array .

A simple example of an addition of two cells is used in this algorithm to show how it works. The reality is more complex, as non-strict expressions like

```
=IF(RAND()<0.5;1;2)
```

exists. Only one of the two branches is going to be recalculated and the algorithm above recalculates that branch depending on the outcome of the test. These details do not affect the running time or complexity, they only take up space when being presented, so they are omitted. Using an array for representing data, makes it possible to represent formulas containing possible cycles. Cycles still pose a problem but using a array the problem only shows up during recalculation and only if the calculation actually enters a cycle. It is thus possible to represent cycles and even perform calculations on a workbook with possible cycles as long as an actual cycle in the recalculation is avoided. Using this depth first calculation strategy in which cell values is calculated exactly once using a cache, while possible visited many extra times, is rather similar to lazy evaluation in a functional language. The drawback of this depth first approach is that the stack usage is linear in the number of nodes between the top node and the maximal tree depth in the dependency graph.

The recalculation time when using an array as data structure, is $O(m)$, that is linear in the number of cells used, but cell updates, insertions and deletions only takes $O(1)$ time.

To keep things simple TinyCalc represents a sheet as a two-dimensional array of references to cells.

4.2.5 Structure of a workbook in TinyCalc

TinyCalc implements workbooks as described in the following list:

- A workbook is a list of sheets.
- A sheet is a two-dimensional array whose elements are either null *or* a cell.
- A cell is either a constant *or* a formula *or* a matrix.
- A constant is either a string *or* a floating point number.
- A matrix can cover many cells and therefore shares a single cached matrix formula between the cells.
- A cached matrix formula consists of a formula, the cell location where the matrix formula was entered and the upper left and lower right cell covered by the matrix.
- A formula is
  - an non-null expression that evaluates to the cell's value
  - *and* a cached value
  - *and* a workbook reference
  - *and* a uptodate field used when recalculating the workbook.
  - *and* a visited field used when recalculating the workbook.
- A value is either a text string *or* a floating-point number *or* a matrix value *or* an error value.
- An expression may be
  - a constant floating-point number
  - *or* a constant text string
  - *or* a cell reference (sheet and a relative/absolute reference)
  - *or* an area reference (sheet and two relative/absolute references)
  - *or* an application of an operator or function to one or more subexpressions.
- Legal expressions follow the grammar in section A.3. It basically is a simple expression grammar as known from most imperative computer languages. Instead of variables though, it has cell references in both A1 and R1C1 style.

4.2.6 Cell references

TinyCalc supports both relative and absolute cell references. It also supports absolute sheet references. While relative sheet references are not hard to support, it has been chosen not to implement these, as it is questionable how useful they are. Written formally, cell references in A1 format follow these grammar rules[3]:

```
A1CellRef        := [Sheetname '!'] ['\$'] ColumnIdentifier ['\$'] RowIdentifier
ColumnIdentifier := 'A' .. 'I' {'A' .. 'Z'}
RowIdentifier    := digit {digit}
```

TinyCalc supports R1C1 style references through this grammar:

```
R1C1CellRef   := [Sheetname '!'] RowR1C1Ref ColR1C1Ref
RowR1C1Ref    := RowR1C1RelRef | RowR1C1AbsRef
ColR1C1Ref    := ColR1C1RelRef | ColR1C1AbsRef
RowR1C1RelRef := R ['[' SignedInt ']']
RowR1C1AbsRef := R UnsignedInt
ColR1C1RelRef := C  ['[' SignedInt ']']
ColR1C1AbsRef := C UnsignedInt
```

As seen in section 4.1.2, R1C1 referencetypes has the desirable property that they remain unchanged when copied. As it is desirable to share as many formulas as possible when doing RTCG, all A1 type references are converted to R1C1 references in TinyCalc.

The internal representation of references should also be unchanged when copied. A class combining both absolute and relative references achieves this.

```
public sealed class RARef {
   //True=absolute, False=relative
   public readonly bool colAbs, rowAbs;
   public readonly int colRef, rowRef;
}
```

It should be noted that by representing both relative and absolute references in this way, relative references need to be converted to absolute indexes before indexing the cells in an array. This imposes a small overhead but it is negligible in contrast to the gain obtained by being able to share formula expressions for RTCG.

4.2.7 Formula grammar

Formula expressions in TinyCalc are given by the BNF grammar in section A.3. Compared to the grammar found in the basis implementation, it has been augmented with:

[3] As TinyCalc uses a single grammar to parse both A1 and R1C1 style references, the `ColumnIdentifier` does not allow `RC` being parsed as a valid A1 column identifier. Hence, the reason that the first char in `ColumnIdentifier` only is allowed to be in the range `'A' .. 'I'`

- Support for references in the R1C1 format.
- Support for string concatenation operator (&).
- Support for numbers in scientific notation.
- Support for sheet references.
- Support for the power operator (ˆ).

4.3 Graphical User Interface

A spreadsheet system comprises of two main parts, a spreadsheet user interface (UI) and a calculation engine. Normally, the UI is graphical (GUI) but could also be text based (TUI) and has been so in the past. For TinyCalc a GU will be implemented.

Focus, when designing the GUI for this system, should be on making it easy and efficient to investigate problems and hypotheses regarding RTCG, but nothing else. Advanced GUI design has very little relevance to the main conclusions of this thesis.

The GUI is implemented in Window's forms, as this framework is part of Visual Studio 2005 for C# in which TinyCalc is implemented. Lengthy discussion about Window forms will be skipped but readers should be aware of that books exists on the topic and that Microsoft already has announced the successor to Windows forms.

With the above criteria in mind, the following list of features and limitations of the GUI are listed:

- Data should be arranged in rows and columns and be easily editable.
- Rows and columns should be named after the normal consensus among spreadsheet systems; that is, rows are numbered using integers, starting from 1. Columns are named using the 26 standard letters in the English alphabet `A-Z`. When using more than 26 columns a "breath first´´ name scheme is used. Exemplified the column names are ordered this way: `A`, `B`, . . ., `Z`, `AA`, `AB`, . . ., `AZ`, `BA`, `BB`, and so forth.
- It should be possible to navigate the GUI using shortcut keys.
- The GUI should be constructed so that access to the whole workbook is easily possible. That is, it should be possible to switch quickly between the individual sheets in the workbook when editing the spreadsheet.
- Visual shortcuts should be implemented. For instance, instead of following the cursor up through the rows to get the actual column name and then following the row to get the row name, the cell name should always be visible a fixed place on the GUI for easy retrieval.

- The following menu items are required to do the most basic things: (create) new workbook, open workbook, close workbook, save workbook, add sheet, add row, add column, delete row, delete column and recalculate workbook.
- Lastly, a configuration form for the spreadsheet should be implemented. This form should provide support for changing global parameters for the spreadsheet.

This is all the system is going to implement with respect to the GUI. In the following subsections, various design choices and problems is presented and the section is concluded with a screenshot of the GUI.

4.3.1 Implementation

Based on the above list the GUI in figure 3 was designed. At the heart of the design are two Windows FORMS controls: the **DataGridView** control and the **TabPage** control.

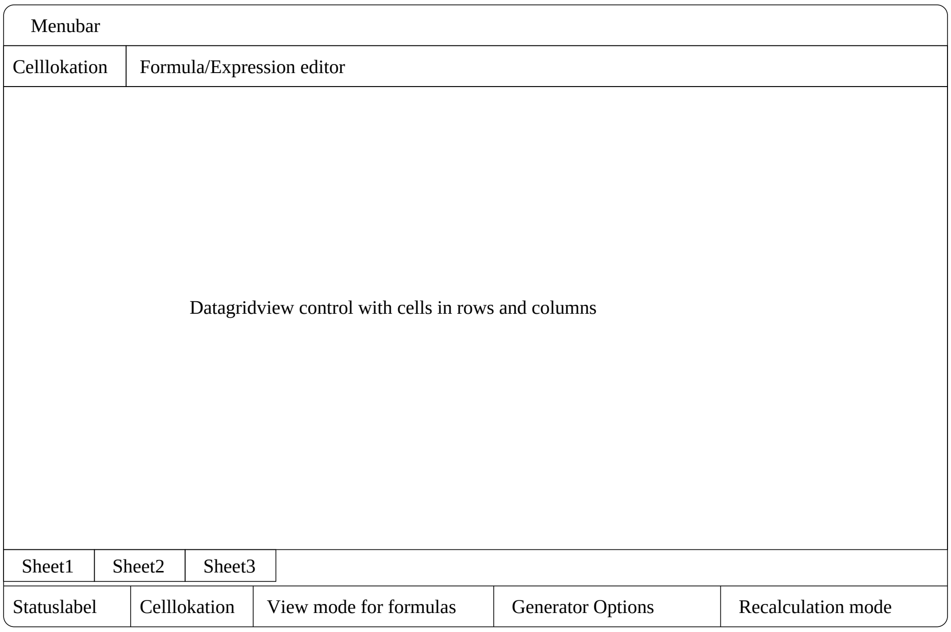
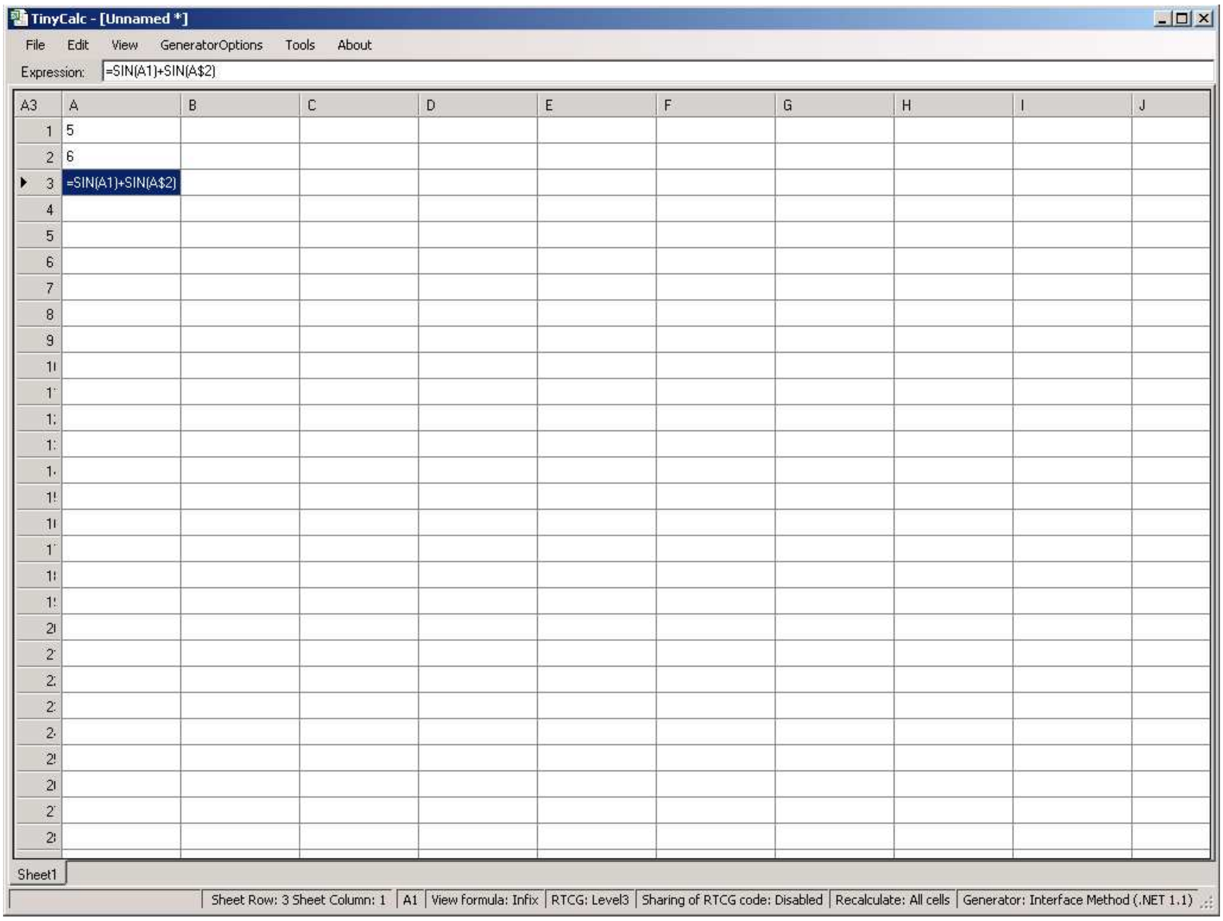


Figure 3: Design of the GUI as it is going to be built.

The **DataGridView** control is capable of displaying strings in a tabular format. It has editing- and layout capabilities, column- and row headers and are able to scroll the grid when needed. It is the perfect control for the job, featurewise.

The **TabPage** control can display other controls in a tab. The user can select which tab to display. It has automatic scrollbar facility which displays a scrollbar if the control contains more TabPages than can be displayed on-screen. A screenshot of the final GUI for the application can be seen in figure 4.



Expression: =SIN(A1)+SIN(A$2)

A3	A	B	C	D	E	F	G	H	I	J
1	5									
2	6									
3	=SIN(A1)+SIN(A$2)									

Sheet Row: 3 Sheet Column: 1 | A1 | View formula: Infix | RTCG: Level3 | Sharing of RTCG code: Disabled | Recalculate: All cells | Generator: Interface Method (.NET 1.1)

Figure 4: Screenshot of the final GUI.

4.3.2 Problems and notes

When building the GUI some problems and annoyances were found. For the record these are presented here.

- The **DataGridView** class is slow. On a 750MHz Pentium III with 192MB RAM, one can follow the screen redrawing when scrolling in spreadsheets with around 1000 cells. The official Microsoft F.A.Q for the DataGridView[3] contains hints and tips on how to reduce the memory footprint of the control or speed it up by avoiding certain usage patterns, but unfortunately none of these hints can be applied to TinyCalc.

- Any attempt to reuse a single **DataGridView** control for *all* the **TabPages** were fruitless. It seems that a **DataGridView** control can have only one parent. The solution to this, in this system, was to use a new **DataGridView** control per tab. This makes the performance and memory problems with **DataGridView** worse, but was by far the fastest way to get the GUI up and running.

4.4 Command line interface

Besides the relative simple GUI, the system also implements a simple command line interface to facilitate scripting of tests and benchmarks. The most important option for the program to implement with regards to the command line interface is the option to suppress the GUI from starting up and just execute a TinyScript (Section 4.5) script. The system is kept simple and help on usage can be obtained by executing `TinyCalc.exe --help`:

```
Usage:

-[v|version]  Shows the current version of TinyCalc
-[h|help] Shows a little help text for CLI usage of TinyCalc
-[s|script][=]<argument> Executes the TinyScript(tm) script given
-[f|scriptfilename][=]<argument> Executes the TinyScript(tm) script
      given by the filename
-[a|scriptargument][=]<argument>  String argument to be given
      to the script. Can be used multiple times.

Arguments can be quoted using doublequotes if needed be.
```

4.5 Support for scripts with TinyScript

To facilitate automated testing, some sort of programming language was needed in TinyCalc. Even though inventing and implementing yet another scripting language is an interesting and educational accomplishment, it is both very time consuming and not especially relevant for this thesis. Instead, it was decided to use C# as the “script” language in TinyCalc and then use the .NET reflection API to compile these C# scripts at runtime.

To do so requires that an API between TinyCalc and TinyScript to be defined. The API is kept simple as TinyScript primarily is a tool for constructing regression tests (section 6) and benchmarks(7.

Large projects normally define dynamic linkable libraries (DLLs) which can be referenced independently in other programs. TinyCalc, however, is build as a stand alone application without dynamic libraries, and while being very modular, the time does not permit a script API being extracted and put into a DLL, which would enable stand alone scripts. Instead, TinyScript defines the methods and classes constituting the TinyScript API among all methods and classes in TinyCalc. At a later time, these methods could be factored out into a DLL. As could the GUI and command line interface, but more about this in section 8.3.

To summarize: TinyScript scripts are in fact C# programs which references methods and classes as defined in an API. These methods and classes live in the TinyCalc namespace.

The TinyScript API consists of classes used to model a spreadsheet, methods performing I/O on spreadsheets, methods to build new spreadsheets from scratch, methods for editing spreadsheets, a single method for performing recalculation, a method for controlling recalculation and formatting of formulas when displaying them. The actual API can be seen in an appendix in section A.6.1.

Two examples of using this API are presented. First, a script that creates a new workbook with two sheets, makes up a couple of formulas, recalculates the workbook and retrieves the results of the calculations. The script can be seen in A.6.2.

The second script creates a new workbook with a single sheet and a couple of formulas, saves this workbook and reopens it as a new instance. It then recalculates both workbooks, one in Level 0 (evaluation) and the other in Level 1 (RTCG for each subexpression) and then compares the results. Outputs the string ”OK” if the calculations are identical. The script can be seen in A.6.3

4.6 Loading and saving spreadsheets

However good a spreadsheet system might be at performing operations on data, it is not a very useful tool if it is time consuming and tedious to get the data in or out of the system. For small data sets, it is possible to use the GUI, but it quickly gets tedious and repetitive if all changes to a spreadsheet are lost when closing the application.

So TinyCalc needs to be able to load and save spreadsheets in some way. Looking at the three spreadsheet systems Excel, OOCalc and Gnumeric it looks like XML is the way to encode spreadsheets and office documents in the years to come. Excel presently (2006) has it own proprietary `.xls` format, but introduced XMLSS in Excel 2002 and have announced a successor to XMLSS called Office Open XML format[17][4] to compete with the OpenOffice standard. This successor is scheduled for ECMA standardization. While `.xls` files are smaller and loads/saves faster, encoding data

[4] The Office Open XML format draft 1.3 consist of approximately 4000 pages!

in XML is actually a good thing. XML is designed to be rigid in structure and easily parsable by computers, while (possibly) still being readable by humans.

Based on these observations, TinyCalc will also store spreadsheets in XML. Instead of inventing yet another XML document format, the three formats used by Excel, OOCalc and Gnumeric will be investigated and a format will be picked for use in TinyCalc. The choice will depend on the following:

1. How readable is the saved data? Put in a different way, how complex is the XML markup in the given format?
2. How well documented is the format and how difficult is it to obtain this documentation?
3. How likely is it that third party data is delivered in this format? How much interoperability does a given format give TinyCalc?
4. Is the encoding single file or multi file and is the datastream compressed?

The overall criteria for selecting a format is ease and speed of implementation.

With regard to 4.) above, it should be noted that while applications might produce a single file having the `.xml` suffix this might be an illusion. Often the file is actually some kind of compressed file or archive, as XML can be quite voluminous and as text it has excellent compression ratio. Furthermore, applications take advantage of the fact that compressed files can contain a directory structure, so the singular `.xml` file is often an illusion for a compressed filed containing a directory with many small files.

When choosing the actual format for use in TinyCalc, we will disregard anything not related to computations. More specifically, data format options, cell format options, sheet layout options, metadata and document history is disregarded. So is a given format's ability to store figures, images and charts. That said, it will briefly be stated if a given format supports these features.

4.6.1 Test sheet for XML investigation

A simple test sheet is constructed in order to investigate how the three systems encode their data. The test sheet is constructed so it uses all the features and types that TinyCalc is going to support. It does also contain some features and types which are not going to be supported by TinyCalc in order to investigate if nonessential features might hinder the usage of a given format in TinyCalc. The test sheet is constructed so that it contains:

- At least two sheets in the workbook.

- A couple of floating point numbers and strings, ie: 5, 6.0, 7.0e0, 8.0e1, “test-string”.
- A couple of special chars and strings, ie is any chars escaped?
- A formula using purely absolute addressing.
- A formula using purely relative addressing.
- A formula using both absolute and relative addressing.
- A formula using absolute intersheet addressing.
- A formula using relative intersheet addressing if the spreadsheet supports it.
- A formula combining both absolute and relative intersheet and absolute and relative formula references.
- Three matrices, two 2x3 and a 1x1, to see how they are encoded. Can a 1x1 matrix be distinguished from a double value?
- A matrix calculation formula.
- A formula using a fixed function, ie the sinus function SIN
- A formula using a dynamic function, ie the random function RAND.
- A couple of types that TinyCalc is not going to support at first: date/time and currency formats.
- A couple of examples of markup such as colors and justification.
- Examples of more exotic features such as weblinks, embedded images, charts and cellcomments.

The three spreadsheets are made as identical as possible and it is then investigated how each spreadsheet system encodes this sheet in their own markup. They can all be found on the accompanying CD-ROM in the `XML analysis` directory. In the same directory there also exist cut-down versions of the test sheets in which the XML markup for the nonessential features has been removed.

4.6.2 Gnumeric Format used by Gnumeric

Gnumeric has developed its own format for storing data. There exists very little documentation about this format. The official documentation [11] consists of a webpage and contains almost no relevant information besides stating that Gnumeric compresses the `.gnumeric` files with `gzip` for space reasons. It does, however, provide a link to [12] which describes the gnumeric format as seen by a developer when implementing support for the gnumeric format in JWorkBook. This document contains 30 pages and together with the test sheet it provides nearly enough information to deduce what is necessary to implement support for the format. Two things are missing. Firstly, a description of defined `ValueType`s is lacking. However, looking at `value.h` in the source[13] for Gnumeric the missing information can be found. Secondly, there is no grammar or textual description of the formula expressions.

Gnumeric stores workbooks in a *single* file, *compressed* with *gzip*. It includes style and layout information for the data values. It includes document history and display options as metadata. It supports *embedding of objects and images* inside the XML file. It has support for named cells and, finally, it conceptually support sharing of identical formulas as it introduces an index for a particular formula and then reuses that index later on if a identical formula should reappear in the spreadsheet.

It should be noted that while it appears that it is possible to support the gnumeric format, the documentation is produced by a third party person and is from 2001, hence nonauthoritative and old. Furthermore the documentation lacks information on whether XML elements and/or attributes are optional or required, making it uncertain how complete and error free any support might be. The format will not provide great interoperability with other spreadsheet systems.

Finally, it is noted that it is not possible to construct a single pass forward only parser for this format without some kind of memory.

4.6.3 XMLSS Format used by Excel

Excel can save in a XML format in addition to its native `.xls` format. This XML format is called XMLSS for XML SpreadSheet. On MSDN there are two webpages, [27] and [26], documenting the XMLSS format. These are not complete in the sense that they cover all details in the format but especially [27] is very thorough. These webpages and the test sheet are enough to implement support for XMLSS in TinyCalc.

An XMLSS file is a *single noncompressed* file. It includes style and layout information for the data values and metadata such as document history and display options. It lacks support for embedded objects and graphs [35]. It supports named cells and named ranges, but has no support for formula sharing in the format.

There is no grammar describing how formula expressions are constructed. There are however a couple of attempts ([4] and [1]) at constructing a BNF grammar for what people *think* constitutes Excel formula expressions. The interoperability with other spreadsheet systems using the XMLSS format are very high either directly or through Excel.

It is not possible to construct a single pass forward only parser for this format without some kind of memory.

4.6.4 ODF Format used by OOCalc 2.0

OOCalc 2.0 uses Open Document Format (ODF) version 1.0, see [25]. The standard is a 706 page large PDF file, organized so that small sections describe distinct features (document metadata, named items, links, images and so forth) and then it is up to an application developer to decide how many of these features the application is going to support. There is, however, a recommended list of points for any given type of application such as a wordprocessor, a spreadsheet system, and so forth. Using this list as a guideline, the standard recommends that for a *spreadsheet application* features that cover about 410 pages is to be considered.

The ODF format is *compressed* using the *zip* format. It contains a *directory* structure. It is optionally *encrypted* as are the files in the directory structure. A *Manifest* file in the root directory provides an entry point to decoding data. This manifest file contains information about files, paths, compression- and encryption methods and finally any keys needed for decryption.

An indented version of the reformatted test sheet called `contents-reformatted.xml` can be found on the CD-ROM, as can a cut down version. There are more than enough information to implement both read and write support for ODF in TinyCalc.

It comes as no surprise that ODF supports almost anything Gnumeric or XMLSS supports. This includes named cells, named ranges, images, charts, layout and style information for data and metadata.

The documentation does not give any grammar for formula expressions in ODF and lastly, it is noted that it is not possible to construct a forward only single pass XML parser for TinyCalc using this format.

4.6.5 Selection of format for TinyCalc

This table lists the features of the three formats relevant for this thesis:

	ODF	XMLSS	Gnumeric
Documentation	Excessive	Plentiful	Incomplete
Documentation status	up to date	up to date	outdated
Dir/file layout	Dir structure	Single file	Single file
Compressed	Yes, Zip	No	Yes, GZip
Encrypted	Possible[5]	No	No
Formula encoding	A1	R1C1	A1
Single pass possible	No	No	No
Named Cells	Yes	Yes	Yes
Charts and Images	Yes, extern	No	Yes, inline
Formula indexation	No	No	Yes
Format styles	Yes	Yes	Yes
Metadata	Yes	Yes	Yes
CellRef Origo in markup[6]	(undef,undef)[7]	(1,1)	(0,0)
Formula grammar	No	3rd Party	in the source

Based on the fact that XMLSS is the simplest format combined with the fact that it is well documented and provides excellent interoperability with Excel and OOCalc as OOCalc can read XMLSS files, whereas Excel cannot read ODF files, XMLSS is chosen as the primary format for TinyCalc.

That said, both ODF and Gnumeric are interesting in a broader sense; ODF in a broader context than this thesis, because it is very likely to become *the* Open Source standard for document encoding in the years to come. It has major momentum going for it in the public service sector as well as in the private sector and is scheduled for ISO standardization. Gnumeric, on the other hand, is interesting in this thesis as it does formula indexing in the Gnumeric format.

Should time permit any further development of input and output formats, the formats are prioritized this way:

1. Read support for Gnumeric.
2. Read support for ODF.
3. Write support for ODF.
4. Write support for Gnumeric.

[5] ODF can be encrypted with a vendor specific method.

[6] The cell references in question are those presented by the markup, not those in the actual formulas.

[7] ODF does not define any absolute indexes in its markup, so it is undecidable.

4.6.6 Implementation

.NET provides many ways to work with XML but only two methods are considered here:

- One possibility is to parse data using a forward only scanner, extracting data as it appears in the stream. This has a speed and memory advantage for large inputs as almost no state is kept during parsing. The disadvantage is that it is hard or impossible to update an existing document or preserve data not relevant for TinyCalc. A forward only scanner works best if it is possible to scan the whole document in a single pass. The **XmlTextReader** class implements a fast forwarding XML scanner in .NET.
- The other possibility is to read in the XML data into an internal XML treestructure. For large inputs, the memory and cpu usage of doing that might not be negligible. It does, however, make it easy to keep structure of the XML file and easily update this structure without touching structure of elements of no importance for our application. An internal XML document is well suited if multiple passes over input are required. The **XmlDocument** class implements an XML scanner with an internal XML representation.

XMLSS can not be parsed in one pass using **XmlTextReader**, but considering the fact that some people recommend **XmlTextReader** over **XmlDocument** when the XML files is large [14], **XmlTextReader** is chosen. Our hypothesis is that one of the areas where RTCG might be beneficial in spreadsheet calculations is on *large inputs* with lots of similar formulas.

Implementation should be straightforward with the exception of the following point. In order to fully exploit RTCG, identical formulas should share the same representation internally. XMLSS does not support formula sharing, but stores the formulas as simple text strings. Therefor as written in section 4.2.2 TinyCalc needs to:

1. Parse the formula from the textstring into an abstract syntax tree using a scannar/parser.
2. Reconstruct the formula as a string using R1C1 style references.
3. Using a dictionary/hash table (for fast lookup) of strings to determine if a formula has been seen before and can therefore be shared.

As a last note, it can be said that time *did* permit read support for Gnumeric to be implemented. It uses a **XmlDocument** class and it can be noted that though **XmlDocument** uses more memory than **XmlTextReader** it appears that the memory requirements are four times the size of the input XML spreadsheet file on average. So most spreadsheets can be read using an **XmlDocument** class. A program library called sharpziplib[19] is used for the GZIP decryption of `.gnumeric` files. This library can also decrypt ZIP files used in the *.ods* format.

The code implementing I/O in TinyCalc is found in the `WorkBookIO.cs` file.

4.7 Localization issues

As COCO/R currently only allow a single generated parser in a namespace, a couple of issues arises. These are caused by the fact that the single parser has to parse formulas entered in the GUI, appearing in TinyScript scripts and as encoded in the XML files. The problem is three fold:

1. Formulas are parsed using the `double.Parse(...)` method. The method is culture based and follows the culture defined by the operating system.
2. The three XML formats encode their formulas using English cultural settings.
3. There is no general consensus among the XML formats as to which token that separate numbers in a list. Some uses ';' whereas others uses ','. This ambiguity causes problems in conjunction with the cultural issues and makes it non-trivial to construct a single parser which can parse formulas from all "input" sources.

The solution to the problem has been to construct the grammar so that it parses formulas as appearing in XMLSS, that is English cultural encoding of formulas. Furthermore it is required that TinyCalc operate under an English cultural setting in the underlying operating system.

5 Augmenting spreadsheets with RTCG

5.1 What is RTCG?

Runtime Code Generation (RTCG) can be used for program specialization. Program specialization is a technique where a specialized version of the original general program is produced, either during runtime or compile time. More formally, consider the program P with two inputs, the static input S and the dynamic input D. When the program P is run with these inputs, it produces the result R, which we write as:

$$P(S, D) = R$$

A specialized program is a program, which takes the dynamic input D as parameter and produces the exact same result $P_S(D) = R$ as the original program.

While it is possible to specialize programs by hand, it is tedious and errorprone. People, however, have done it for years. Some types of code optimization can be seen as program specialization. There is a subtle point, however. People skilled in the art of writing hand optimized code, often do more than write specialized programs as defined above. They often also impose assumptions on the original program which enable them to further optimize the code. The original program and the specialized program will still produce identical results in the common case, but the error cases might be different.

Instead of specializing programs by hand, a program that automates the task can be written. Such a program is called a *partial evaluator* and is essentially a program or generating extension which produces programs as output.

It is possible to specialize programs both at compile time and at runtime. Compile time specialization *eliminates* the cost of performing specialization at runtime. On the other hand, compile time specialization is hard, as it is hard to deduce which input is static and which is dynamic at compile time. Furthermore, a runtime specializer is smaller and easier to develop than a compile time specializer.

Given that compile time specialization is hard and time consuming, runtime code generation will be used in TinyCalc through a generating extension. This requires that the overhead of invoking the code generator is minimized. Section 5.5.1 examines the overhead of doing RTCG.

5.2 Possibilities for RTCG in a spreadsheet

The possibilities for RTCG in a spreadsheet are plentiful and can be graduated according to how many CPU and memory resources are used in the RTCG phase.

To support various optimization strategies an optimization level hierarchy is defined in TinyCalc. Level 0 does not generate any code at runtime but uses interpretation to evaluate expressions. Level 1 and all subsequent levels generate code at run time. The hierarchy looks like:

Optimization level	Description
Level 0	Evaluation of formula expression by interpretation as performed in the initial system delivered by Peter Sestoft.
Level 1	A piece of code is generated for each subexpression, that is: `A1+A2` generates 3 pieces of code which needs to be evaluated. One for referencing `A1`, one for referencing `A2` and one that applies the `+` operator on the referenced values.
Level 2	At this level, inlining of subexpressions is performed so only one piece of code is generated for each formula expression.
Level 3	Level 3 does inlining like Level 2 and also subexpression type analysis in an attempt to remove runtime type checks.
Level 4	As Level 3, and also avoids creating intermediate **Value** objects if possible
Level 5	As Level 4, and also inlines constant values if possible.
Level 6	As Level 5, and avoids generating `stloc` and `ldloc` in the code, optimizing the best case as much as possible.
Level 7	As Level 5 (NB!), while extending the scope of the type deduction to cell references.
Level 8	As Level 7 with value optimizations in the abstract stack machine in order to remove dead branches in `IF` statements.
Level 9	As Level 8 but also specializes spreadsheet functions.

The individual levels will be described in greater details in subsequent sections. The analysis will start at Level 1 and use C# as a pseudo language. References will be made to the actual IL code generated. Four simple examples will be used, each being good at making a point at various levels. Of course the optimizations apply to all expression constructs, but some benefit more than others from level to level. The Four examples as formulas

$$\mathrm{A1} = \mathrm{A2+A3} \tag{11}$$

$$\mathrm{A1} = \mathrm{A2+A3+A4} \tag{12}$$

$$\mathrm{A1} = 5+6 \tag{13}$$

$$\mathrm{A1} = 5+6+7 \tag{14}$$

In the discussion that follows, it is assumed that the **Expr**, **Value** and **NumberValue** have the definitions below.

```
public abstract class Expr { ... }

// A Const is a constant (immutable, sharable) expression
abstract class Const : Expr { ... }

class NumberConst : Const {
   private readonly NumberValue value;
   ...
}

public abstract class Value { }

// A NumberValue is a floating-point number
public class NumberValue : Value
{
   public readonly double value;
   ...
}
```

In order to see the differences between the different optimization strategies, IL code for two of the four examples (`A1=A2+A3+A4` and `A1=5+6+7`) has been put in appendix A.5 and can it is recommended to have ready when reading the following sections.

5.2.1 RTCG Level 0 - Interpretation

At this level, no code is generated at runtime. All expressions are *interpreted*, using the `Eval` method implemented by each (sub)expression. The algorithm presented in section 4.2.4 is used to recalculate the entire workbook.

5.2.2 RTCG Level 1 - Distinct RTCG for subexpression

At Level 1 code is generated at runtime. A piece of code is generated for each subexpression corresponding to the fact that each subexpression implements an `Eval` method. When a piece of code for an expression is evaluated and that expression depends on (sub)expressions, these are in turn evaluated by function calls. Consider the formula expression (11). At Level 1, three pieces of code are generated, two implementing cell references to cells `A2` and `A3` and one for the addition which calls the two other methods to obtain the values of the cell references. Under the assumption that the subexpressions `A2` and `A3` are located in the expression array `es`, the generated function will implement this `Eval` function:

Example 4 Example of pseudocode to calculate A2+A3 at Level1

```
Value Eval(Sheet sheet, Expr[] es, int col, int row) {
   if (es.Length == 2)
   {
      NumberValue v0 = es[0].Eval(sheet, col, row) as NumberValue,
                  v1 = es[1].Eval(sheet, col, row) as NumberValue;
      if (v0 != null && v1 != null)
         return new NumberValue(v0.value + v1.value);
      else
         return new ErrorValue("ARGTYPE");
   }
   else
      return new ErrorValue("ARGCOUNT");
}
```

5.2.3 RTCG Level 2 - Inlining

Level 2 tries to inline as much code as possible so that only one `Eval` method is generated for each formula expression. Considering formula expression (11) again, the generated function implements this `Eval` method:

Example 5 Example of pseudocode to calculate A2+A3 at Level2

```
Value Eval(Sheet sheet, int col, int row) {
   if (es[0].sheet != null)
      sheet = es[0].sheet;
   CellAddr ca = es[0].raref.Addr(col, row);
   Cell cell = sheet[ca];
   NumberValue v0 = cell == null ? null :
               cell.Eval(sheet, ca.col, ca.row) as NumberValue;

   if (es[1].sheet != null)
      sheet = es[1].sheet;
   CellAddr ca = es[1].raref.Addr(col, row);
```

```
      Cell cell = sheet[ca];
      NumberValue v1 = cell == null ? null :
                  cell.Eval(sheet, ca.col, ca.row) as NumberValue;
      if (v0 != null && v1 != null)
         return new NumberValue(v0.value + v1.value);
      else
         return new ErrorValue("ARGTYPE");
   }
```

Note that the `Eval` method does not require a subexpression array **Expr[] es** — all the subexpressions have been inlined and the `es` array appearing inside the function represents the expressions at compile time! Secondly note that the calls to `raref.Addr` also can be inlined, but has been omitted in this example. Thirdly note that the check for the right number of subexpressions are performed when generating code[8]. Lastly note, that while two calls to subexpressions can be omitted, the call to the *cells* `Eval` method cannot!

Inlining of `Addr` method call in the **RARef** class is possible as written above. This call does the following:

Example 6 `Addr` method in the **RARef** class

```
   // Absolute address of ref
   public CellAddr Addr(int col, int row) {
      return new CellAddr(this, col, row);
   }
```

The call to the constructor for **CellAddr** performs:

Example 7 **CellAddr** constructor

```
   public CellAddr(RARef cr, int col, int row) {
      this.col = cr.colAbs ? cr.colRef : cr.colRef + col;
      this.row = cr.rowAbs ? cr.rowRef : cr.rowRef + row;
   }
```

Clearly these two small code snippets can be inlined as well.

5.2.4 RTCG Level 3 - Type check removal

At Level 1, no optimization are performed. At Level 2, the optimizations are: call inlining and removal of the argument check. At Level 3, typechecks of subexpressions are performed at compile time. This is done using a simple abstract stack machine.

[8] In case the generator detects a wrong number of parameters it will generate specialized code which returns an **ErrorValue**.

The code generator will use the abstract stack machine to deduce the type of values returned by subexpressions and avoid the typecheck and just do the calculation. If the types are not compatible with the operation, or can not be deduced at compile time the generator will instead generate code that will catch the type errors at runtime and return an `ErrorValue` of `ARGTYPE` as in the previous levels. That is, for `A1 = 5 + 6` the generator would deduce that

$$\mathbf{NumberValue} + \mathbf{NumberValue} \mapsto \mathbf{NumberValue}$$

and then generate what corresponds to:

Example 8 Example of pseudocode to calculate 5+6 at Level3

```
Value Eval(Sheet sheet int col, int row) {
   NumberValue v0 = es[0].Value;
   NumberValue v1 = es[1].Value;
   return new NumberValue(v0.value + v1.value);
}
```

This code saves the tests that checks that v0 and v1 really are **NumberValues** as they are supposed to be. Note that this code also is a lot simpler than the previous code examples because we changed from cell references to constant values, a change due to the fact that at Level 3 TinyCalc only performs subexpression type optimization, and the point can not be made with an example using cell references.

When generating code for a single formula expression at a time, types inside the expression cannot change at runtime. Types of values in referenced cells on the other hand depend on the values in the cells, and these can change at runtime and/or between subsequent evaluations. But more about this in section 5.2.8.

5.2.5 RTCG Level 4 - Avoid intermediate Value's

Level 4 tries to avoid generation of intermediate **Value** objects. Consider the expression `A1=5+6+7`. At Level 3, the generated code implements a function that looks like:

Example 9 `A1=5+6+7` at Level 3

```
Value Eval(Sheet sheet int col, int row) {
   NumberValue v0 = es[0].value;
   NumberValue v1 = es[1].value;
   NumberValue v2 = new NumberValue(v0.value + v1.value);
   NumberValue v3 = es[2].value;
   return new NumberValue(v2.value + v3.value);
}
```

However this can be shortened. There is no reason to construct **v2**. The only thing it is used for is a placeholder for the final addition. An improved generator, would instead deduce that the **v2** is not used for anything except being a placeholder and then generate:

Example 10 Avoiding `new NumberValue(...)` in `A1=5+6+7`

```
Value Eval(Sheet sheet int col, int row) {
   NumberValue v0 = es[0].Value;
   NumberValue v1 = es[1].Value;
   double v2d = v0.value + v1.value;
   NumberValue v3 = es[2].Value;
   return new NumberValue(v2d + v3.value);
}
```

5.2.6 RTCG Level 5 - Embed constants in IL code

At Level 4, new objects were avoided by using locals (of type **double** and **string**) instead of **NumberValue** and **TextValue**'s. But why stop there? Constants have constant types and, as such, an even better generator would generate IL code that implemented the following C# function for `A1=5+6+7`.

Example 11 Calculating directly in local variables

```
Value Eval(Sheet sheet, int col, int row) {
   double v0d = es[0].Value.value;
   double v1d = es[1].Value.value;
   double v2d = v0d + v1d;
   double v3d = es[2].Value.value;
   return new NumberValue(v2d + v3d);
}
```

The above example is a bit contrived, as it is not very likely that three constants are added together to form a new constant in an actual spreadsheet, but the point is that constants can easily be type deduced and optimized when generating code.

5.2.7 RTCG Level 6 - CLR calculation

Considering the IL code for the example of `A1=5+6+7` at Level 5, it can be seen that code uses `ldloc` and `stloc` quite a bit. At Level 6, the generator tries to avoid using these constructs and *simply* leave the actual **double** or **string** objects on the stack of the CLR, when the compile time type checks have verified the expression, it can be done safely.

5.2.8 RTCG Level 7 - Speculative type deduction

The type deduction optimizations introduced in Level 3 are limited by the fact that they only can be applied to subexpressions of formula expression. It would be preferable to do type analysis across cells, so values obtained from cell references could be part of the type analysis. This analysis will be called speculative type deduction in the future and in the next sections it will be clear why.

There are difficulties with letting the type analysis span cell boundaries. Consider a situation where `A2=8, A3=9` and `A1=A2+A3`. With speculative type deduction, it is assumed that `A2` and `A3` both hold a **NumberValue** and this assumption allows the codegenerator to skip typechecks from the generated code. Now consider what happens if the user changes cell `A2` from 8 to "Thomas"? Instead of `A1` containing a **NumberValue** of value 17 it should now hold an **ErrorValue** of value `ARGTYPE`. But what had happened if the user changed `A2` from 8 to another **NumberValue**, for instance 10? Nothing! At least not with regard to the codegenerator and the generated code for `A1`. The assumption that `A2` and `A3` is of **NumberValue** type still holds. This leads to two interesting points:

1. Deploying speculative, intercell type analysis requires dependency information from referred cells and formula expressions to the referring expression. That is, when changing the content of `A2` or `A3` above, the system needs to know that `A1` depends on the (types) of these cells.
2. Only when changing a referred cell to a value or expression of *another type* will the referring expression need to be recompiled.

5.2.9 RTCG Level 8 - Value optimizations

Optimization Level 8 extends the deduction of types to cover also constant values. Doing value analysis at compile time will allow the generator to skip dead branches, most notably `IF(Test;Branch1;Branch2)` constructs. Not only will the generated code be shorter, but it might be possible to avoid using time on generating code which never gets executed. It is, however, questionable whether this optimization level is worthwhile doing in general.

5.2.10 RTCG Level 9 - Function specialization

At this level, spreadsheet functions are specialized with regards to static arguments. It should be noted, that not all functions are good candidates for specializing. Some can not be specialized at all, some only in special circumstances, and finally some functions requires a great deal of cpu time to be specialized. This makes it unclear whether it is worth the effort to specialize functions in TinyCalc.

5.3 Imposed limits on RTCG in TinyCalc

Due to time constraints only RTCG Level 0-6 are implemented in TinyCalc. Level 7 is implemented with a presumed working type deduction scheme. An actual type deduction using dependency information is not possible due to time constraints, but by assuming that the infrastructure exists, Level 7 with type deduction on cell references can be implemented in the code generator in a very short time. This allows the potential of doing type deduction on cell references to be evaluated and it can be concluded whether it is worth the effort to implement it fully in a later project.

RTCG support for **MatrixValue** is also implemented, but is not as complete as for **NumberValue** and **TextValue**'s. As matrices poses a new set of optimization problems and solutions, a new set of optimization levels has been defined (Section 7.4).

Lastly it is noted that all optimizations described in the previous sections also is used only if they can be applied to some of the subexpression. For instance `A1 = 5 + A2`, where `A2=6` will at Level 3-6 still produce optimized code for the **NumberValue(5)** subexpression even though the `A2` subexpression first can be optimized at Level 7.

5.4 Sharing of RTCG for formula expressions

As seen in section 4.1.1, implementing formula expressions in a smart way implies that equivalent formulas can share the *same* formula expression. When combined with RTCG, this opens up the possibility of performing the RTCG once for each shared formula expression. A spreadsheet can be expected to contain many identical formulas thereby reducing the penalty of performing RTCG on the spreadsheet.

The code generated should be bound to the expression being used for the generation. This means that as long as the codegenerator does not assume anything about types and/or values of other cells, code sharing is easy (Level 1-6). Sharing of the generated code for identical formula expressions, where the codegenerator tries to be speculative about types and/or values in other cells, however, is not that easy. A couple of examples will clarify these matters. Consider:

$$\texttt{A1} = \texttt{B1} + \texttt{C1}\ (= \texttt{RC[+1]+RC[+2]})$$

Copying A1 to A2 yields:

$$\texttt{A2} = \texttt{B2} + \texttt{C2}\ (= \texttt{RC[+1]+RC[+2]})$$

as expected. Obviously, code for the formula expression RC[+1]+RC[+2] can be generated once and used twice. At least this is the case as long as the generator method does not do speculative type deduction as defined in section 5.2.5. Consider an alternative expression:

$$\texttt{A1} = \texttt{"foo"} + 8$$

Clearly this should yield #ERR:ARGTYPE as a value. As both "foo" and 8 are expression constants, non-speculative type deduction is safe, and the final code can be boiled down to a couple of IL instructions that just return the #ERR:ARGTYPE value. Clearly, the IL code implementing the expression can be shared when copied to A2. Now consider what happens at Level 7 with speculative type deduction in the following situation:

$$\begin{aligned}\texttt{A1} &= \texttt{B1} + \texttt{C1}\ (\texttt{RC[+1]+RC[+2]})\\ \texttt{A2} &= \texttt{B2} + \texttt{C2}\ (\texttt{RC[+1]+RC[+2]})\\ \texttt{B1} &= \texttt{B2} = \texttt{"foo"}\\ \texttt{C1} &= \texttt{C2} = 8\end{aligned}$$

The generator deduces that B1=B2 and C1=C2 contains different types and generates IL code which just returns #ERR:ARGTYPE. Furthermore, the generator deduces that A1=A2 and that the types are still different, so the code can be shared among A1 and A2. Now what happens when the user changes B2 from "foo" to a numerical value, 4 for instance? Both A1 and A2 shows #ERR:ARGTYPE, which is only correct for A1. Cell A2 should really contain the value 12. It could be worse than this. Assume that B1, B2, C1 and C2 all contained numerical values. The speculative generator would correctly deduce that the types were indeed identical and generate code without type checks. If the user later changes, for instance, B2 to a string value, A2 would not evaluate to a #ERR:ARGTYPE as it should, but the CLR would instead generate an exception as the IL stream tries to add a string to a float.

So to summarize, sharing of generated code is straightforward at Level 1 through 6 as the code generator at these levels does not try do deduce anything about the values or types of other expressions. This means that a single piece of code shared by two identical formula expressions *will* behave identically. When sharing code at Level 7+ (speculative deduction), care should be taken to keep track of assumptions about types and values, and generate new IL code that behaves correctly if the assumptions for the current IL code breaks.

5.5 Implementing RTCG in TinyCalc

5.5.1 Evaluation of call overhead of doing RTCG in .NET

There are essentially two approaches to runtime code generation in C# in use today. The first approach generates C# code and the second generates IL code. In the following, the general power function `pow(x,y)` will be used to measure the cost of doing RTCG using various APIs in C#. C# and ILASM versions of these two functions can be seen in appendix A.4.1 and A.4.2.

Generation of C# code can be done at runtime by emitting C# code constructs to a stream, invoking the compiler through external system calls thereby generating bytecode, and finally load the bytecode into the common language runtime (CLR), where the code is ready to be run. A variant of this technique is to use the Code Document Object Model (CodeDOM) in .NET to represent the C# program instead of using a stream. CodeDOM is basically just an abstract syntax representation of code and allows for creating code at runtime using classes and types found in the CodeDOM. After having expressed a C# program as a CodeDOM object, the code can easily be emitted, compiled and invoked. Lastly, it should be noted that the CodeDOM framework also supports compilation and parsing of C# code expressed as literal text. A high overhead of generating code at runtime using C# is to be expected, as the compiler has to be invoked.

In contrast to invoking a compiler, it is possible to emit IL code directly. This removes the cost of invoking the compiler at the cost of making it harder, more time consuming and more error prone to generate the generating extension. Luckily this only has to be done once. On the other hand, expressing the code in ILASM provides greater control over the actual code executing in the CLR, thereby potentially allowing for more specialized and faster code. There essentially exist two ways of generating IL code in .NET 2.0.

- The first method is to use the `DynamicMethod` class and invoke the generated method through a delegate.
- The second method is to build the module from the ground creating an assembly.

In that assembly creating a module and in that module creating an class and in that class creating a method, for that method obtaining an `ILGenerator` finally being able to emit IL code for the body of the generated method.

Figure 5 shows the time it takes to compile code using the various methods above. The benchmark can be found on the CD-ROM in the `OverheadBenchmark` directory along with the actual output (`RTCG_Overhead_Compile.xml`). As expected, it can be seen that invoking the compiler is a costly affair in contrast to emitting the IL code directly. The high overhead combined with a very high level approach rules out C# as the RTCG language.

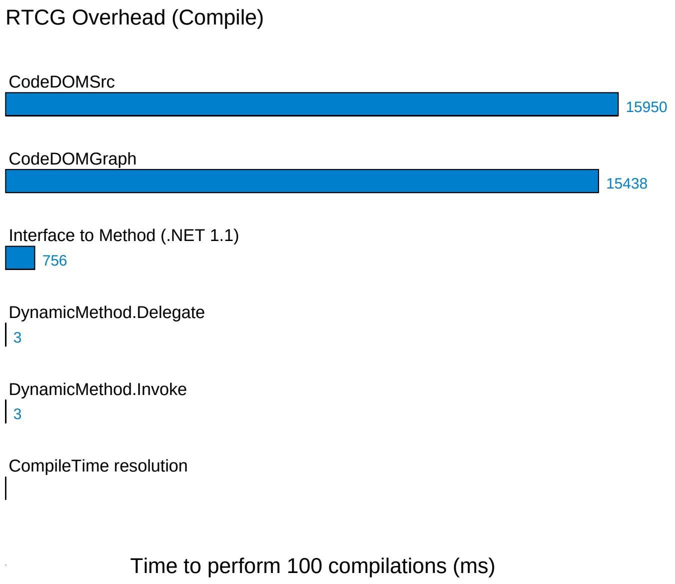


Figure 5: Overhead of doing compilation

It should be noted that the difference between using the interface method and a DynamicMethod is a lot higher than expected, as both methods emit code using an **ILGenerator**. What causes this discrepancy is investigated in the next sections. However, before that, it is noted that, in any case, RTCG in TinyCalc is performed by emitting a stream of IL instructions, and then call a method which executes this stream of instructions. As described by Peter Sestoft in [33] (2002, .NET 1.1) and again by Joel Pobar in [18] (2005, .NET beta 2), it appears that the fastest way to call ILasm code at runtime is through an interface call. As good practise, a simple reevaluation of this fact is performed as things easily could have changed from .NET beta 2 (2005) to the finalized .NET 2 edition (2006) used in this Thesis. More specifically it will be investigated:

1. How long it takes to invoke a static compiled method through compile time resolution.
2. How long it takes to invoke code compiled using CodeDOM (for the reference).
3. How long it takes to invoke a dynamically compiled method through reflection, that is the possibility of a program being able to inspect, alter and invoke its own classes and methods.
4. How long it takes to invoke a static method through a delegate.
5. How long it takes to invoke a DynamicMethod method through a delegate which is the preferred way of doing RTCG i .NET 2.x according to Microsoft.
6. How long it takes to invoke a dynamic method through an interface call which is the .NET 1.1 way of doing RTCG.

The code implementing these tests can be found in the `OverheadBenchmark` on the CD-ROM. The results can be found in the `RTCG_Overhead_Invoke.xml` file. Figure 6 depicts the test graphically.

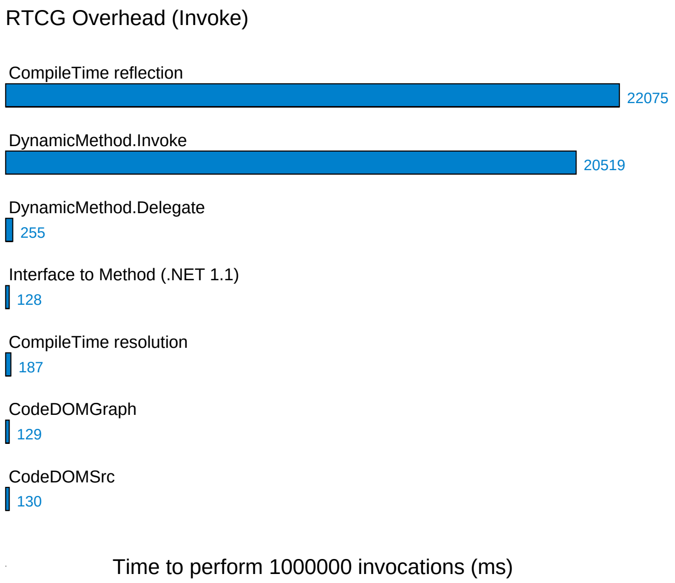


Figure 6: Overhead of doing invocation

The illustration shows that the fastest way to call a method is through static compile time resolution. This is also the case when using the CodeDOM API. It is slowest to use reflection, then a DynamicMethod and last calling a function through a declared interface. It is interesting to note that the fastest method for executing a runtime generated method in 2006 still is through an interface call. This was also the case when Peter Sestoft first investigated these matters back in 2002, so it appears that performance-wise not much has happened in .NET with regards to doing RTCG in the last four years! It is however, claimed [10], that the Dynamic Method should be as fast as the interface method in .NET 2.0, but this has not been the results obtained in this thesis. It is evident, that the stricter a "contract" between the caller and the callee can be formed with regard to parameter passing, the faster a method call can be performed.

5.5.2 Thorough investigation of the interface method (.NET 1.1)

Figure 5 indicated that the overhead of emitting IL code using the .NET 1.1 (Interface) method is slower than the .NET 2.0 (DynamicMethod) method. Taking a closer look at the actual output (`RTCG_Overhead_Compile.xml`), it can be seen that it takes longer and longer time to compile functions using the interface method. To investigate this further, a new test was conducted. This test basically just compiles even more functions in order to reveal the tendency behind the numbers.

The raw output can be seen in the directory `OverheadBenchmarks` in the file `CompileRefInterfaceReuse_Run1.txt`. Figure 7 shows that plotting the total time used compiling as a function of the number n, resembles a quadratic function. It is speculated that this is caused by the fact that each function generated using an interface constructs a new class in which to implement the function. It is likely that it is related to all these classes and that one (or more) structures used by the class loader, shows linear running time in the number of classes when adding an item. A brief test was conducted to test whether the behavior was related to the class names in the example, as they follow a naming scheme of `class+number` (worst case for insertion in a list which maintains a sorted order). The conclusion is: no matter what name the classes are given, the quadratic runtime is seen. So it is simply the sheer number of classes generated that causes this runtime behavior!

5.5.3 Implementation Overview

The implementation of RTCG in TinyCalc are straight forward. There are a couple of points worth mentioning and this section will give a brief overview of the

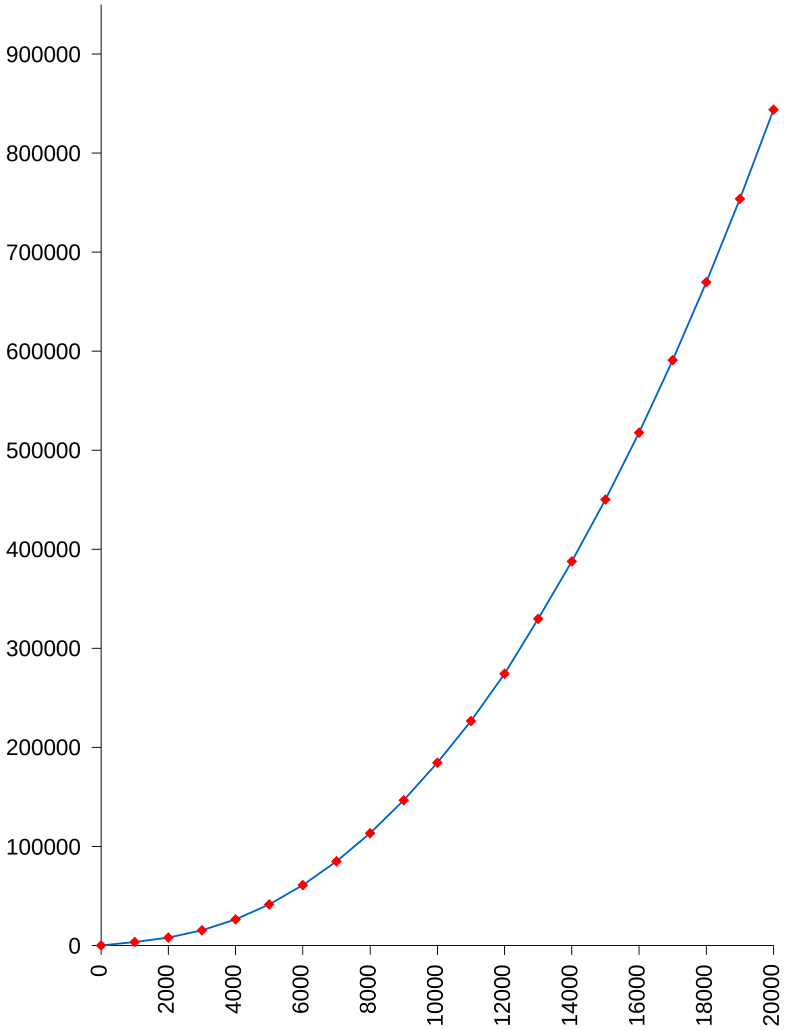


Figure 7: Total compile time (ms) as function of number of compiled functions

implementation.

First it is noted that the underlying abstract machine in the CLR of .NET is stack based. IL assembly instructions first pops its arguments off the evaluation stack *then* performs the operation before the result is pushed back onto the stack. This is contrary to how assembly code in modern native cpus work with registers.

The fact that the abstract machine in CLR is stack based implies that evaluation of formula expressions in TinyCalc needs to be performed in infix order or reverse polish style known from the HP calculators produced by Hewlett Packard. Fortunately, the abstract syntax tree produced by the COCO/R parser makes this trivial. Figure

8 shows an example of a formula expression being converted into an abstract syntax tree. The possible types of expressions were presented in section 4.2.5. Section A.10 shows the possible expression types as class diagrams.

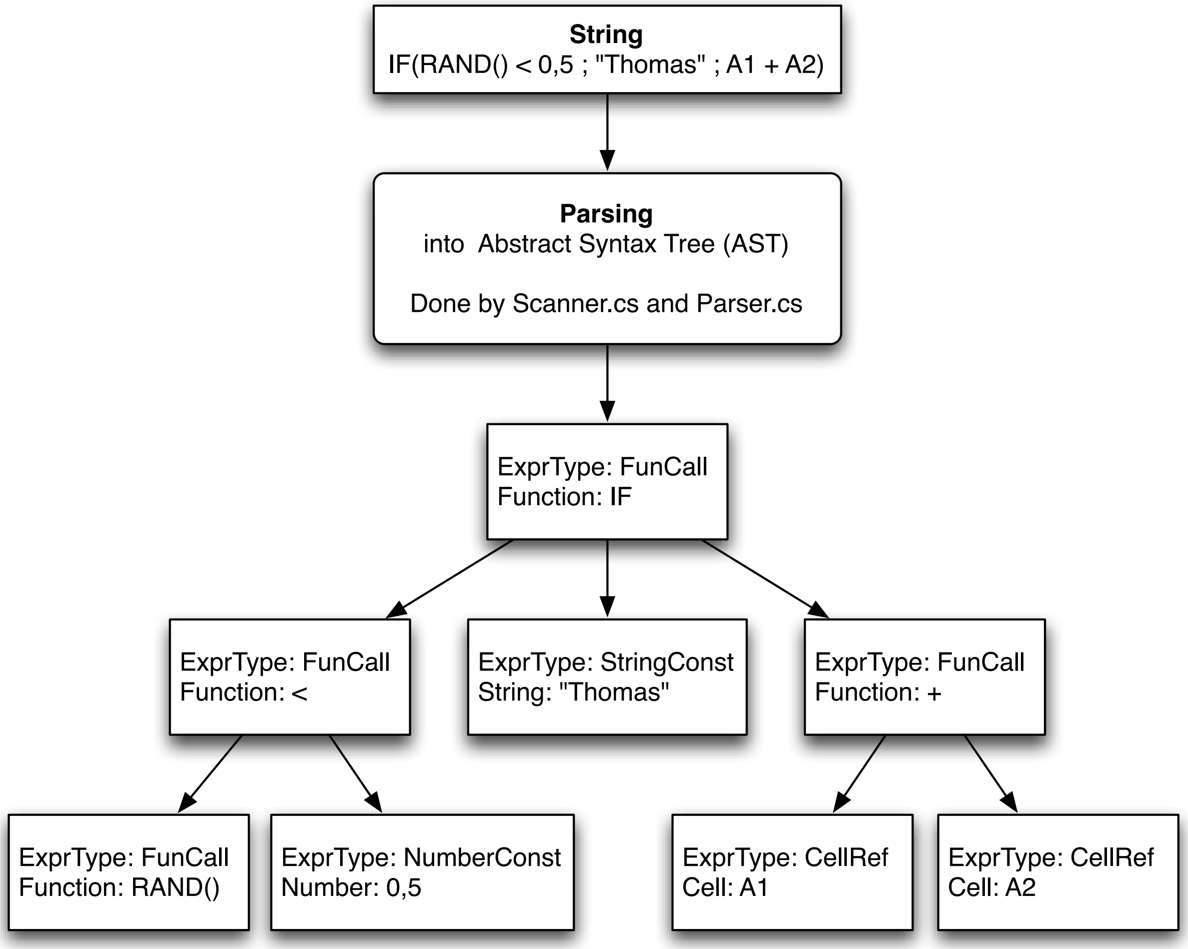


Figure 8: Formula expression parsed into an abstract syntax tree representation.

The main recalculation loop follows the algorithms presented for an array based implement in section 4.2.4. Pseudocode for the recalculation can be found in section A.2.

RTCG is implemented through a generating extension. The original code written by Peter Sestoft implements an

```
Eval(Sheet sheet, int col, int row)
```

method for each expression class. Expression classes can be found in `DOM/Expr.cs`. The **FunCall** expression class is treated specially inside `DOM/Function.cs` due to the high number of functions. Each expression class is augmented with a

```
void Generate(ILGenerator ilg, RTCGExprFieldInfo fii, RTCGAM rtcgam)
```

method for RTCG. Due to the high number of functions, RTCG code for functions can be found in `RTCG/RTCGFunction.cs`. The parameters of the `Generate` method are as follows. The first parameter is an **ILGenerator** used for emitting the actual IL code implementing the expression. The second parameter is an **RTCGExprFieldInfo** containing **FieldInfo** references to fields needed when generating IL for this expression. This class are further discussed in section 5.5.6. The last parameter is an instance of the **RTCGAM** class, which are a Abstract Machine used for type analysis when doing RTCG. This class is further discussed in section 5.5.4. As these `Generate` methods mostly consist of static ilg.Emit statements the "compilation" overhead is minimal.

A simplified layout of TinyCalc can be seen in figure 9. The arrows denotes how information flows towards the CLR. The filenames denotes which files implement that part of TinyCalc. For a full overview of files and directories in TinyCalc see section A.11.

5.5.4 RTCGAM — type analysis

Compile time type analysis is performed with the **RTCGAM** class, implementing a simple Abstract Machine for RTCG. Type deduction is performed using a *type hierarchy* as each expression produces a **Value** of a specific type. From Level 3 and onwards the `Generate` method are required to adhere to a couple of invariants when generating code. Consider an expression e having n subexpressions $e_1 \ldots e_n$. At Level 3 the `generate` method for e is required to:

- *pop* type information off the type stack in the **RTCGAM** class for each of the n subexpressions *after* the subexpressions generate method has been called.
- *push* type information onto the type stack for the type of the return value.

As an example consider the expression $5 + 3$. It consists of two subexpressions: **NumberConst(5)** and **NumberConst(3)**. At optimization Level 2 the `generate` method for + just performs two calls to the subexpressions `generate` methods and then performs the addition:

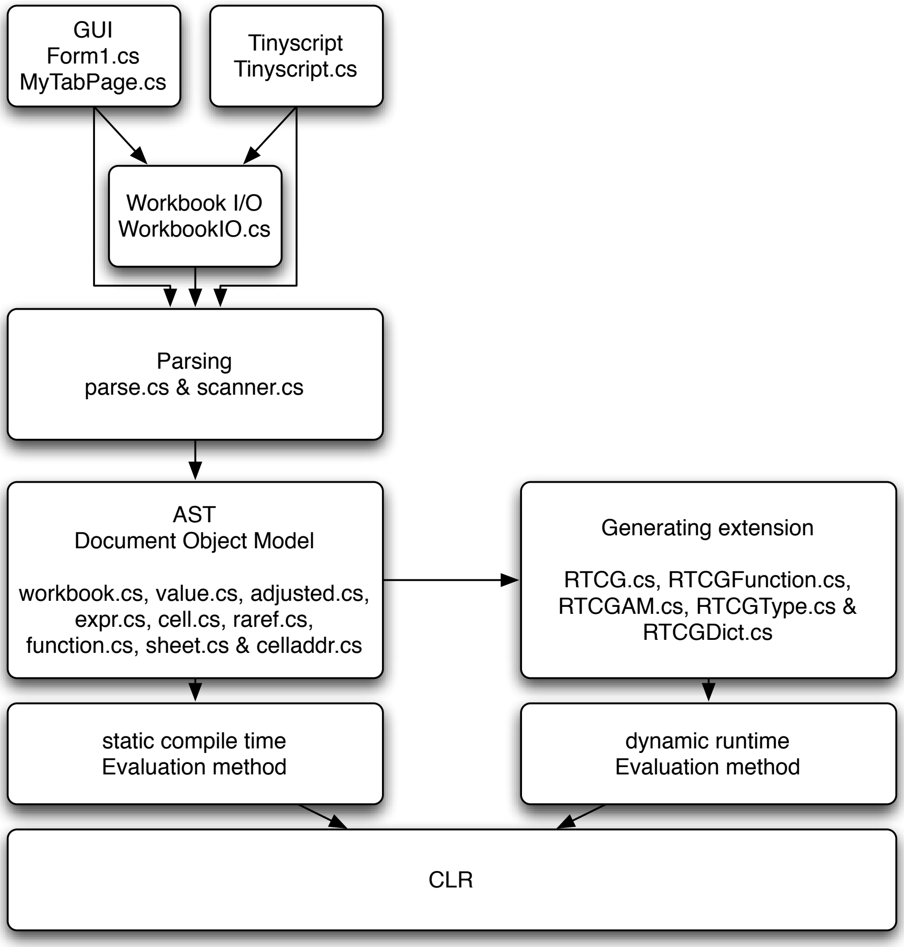


Figure 9: A simplified layout of TinyCalc

```
+.Generate([NumberConst(5),NumberConst(3)]) {
   call 5.Generate(...)
   call 3.Generate(...)
   Generate code for + on 2 NumberValues
}
```

The `Generate` method for + generates IL code with type checks as it is not known before runtime at Level 2, what actually appears on the value stack. At Level 3 the `generate` method for + instead do:

```
+.Generate([NumberConst(5),NumberConst(3)]) {
   call 5.Generate(...)
   {
       Emit_code for NumberValue(5)
```

```
        rtcgam.push(NumberType)
    }

    call 3.Generate(...)
    {
        Emit_code for NumberValue(5)
        rtcgam.push(NumberType)
    }

    t1 = rtcgam.pop();
    t2 = rtcgam.pop();

    if(t1 == NumberType || t2 == NumberType)
       // skip type one or both checks in IL code
       Generate Optimized Code for +
    else
       Generate Normal Code for +

    rtcgam.push(NumberType);
}
```

The type hierarchy are defined in `RTCGType.cs` and looks like:

```
ExprType
   AnyType
      NumberType
         DoubleType
      TextType
         StringType
      MatrixType
      ErrorType
         ErrorArgCountType
         ErrorArgTypeType
```

From Level 5 and onwards (Embed constants in IL code) the **RTCGAM** class is also used in an attempt to leave *double* and *string* values on the CLR stack. For that it is augmented with a stack holding information about which value types an expression *prefer* being left on the stack by a subexpression. Two new rules are added to the invariants for Level 5 to accommodate this. At Level 5 the generate method for e is required to:

- *push* information about which value type it would be prefer the subexpression leaves on the value stack. The push is required to be executed before the subexpressions `generate` method is called.
- *pop* type information off the type stack in the **RTCGAM** class for each of the n subexpressions *after* the subexpressions `generate` method has been called.

- *pop* type information off the preferred type stack in the **RTCGAM** class trying to adhere to what type of value the parent expression prefers being left on the value stack if it is possible.
- *push* type information onto the type stack for the type of the return value.

As an example consider the expression 5+3 again.

```
+.Generate([NumberConst(5),NumberConst(3)]) {

   rtcgam.preferpush(DoubleType)
   call 5.Generate(...) {
      pt = rtcgam.preferpop()

      if(pt can be satisfied)
         Emit code for leaving double(5) on the stack
         rtcgam.push(DoubleType)
      else
         Emit code for leaving NumberValue(5) on the stack
         rtcgam.push(NumberType)
   }

   rtcgam.preferpush(DoubleType)
   call 3.Generate(...) {
      pt = rtcgam.preferpop()

      if(pt can be satisfied)
         Emit code for leaving double(3) on the stack
         rtcgam.push(DoubleType)
      else
         Emit code for leaving NumberValue(3) on the stack
         rtcgam.push(NumberType)
   }
   t1 = rtcgam.pop();
   t2 = rtcgam.pop();

   ptself = rtcgam.preferpop();
   if(t1 == DoubleType && t2 == DoubleType && ptself == DoubleType) {
      Emit(Opcodes.Add);
      rtcgam.push(DoubleType);
   else
      if(t1 == NumberType || t2 == NumberType)
           Generate optimized code without
           typechecks for one or both operands
      else
           Generate Normal Code for + with runtime type check
           for both operands.
      rtcgam.push(NumberType);
   }
}
```

5.5.5 Generating `Eval` methods — parameter differences.

As seen in section 5.5.1 and 5.5.2 it is hard to say if RTCG in TinyCalc in the future uses Dynamic Methods or Interface Methods for its implementation. As written, TinyCalc are going to support both methods as they both are generated through an **ILGenerator** class. There are however, subtle differences between the two methods and in this section these will be described.

The Interface Method builds a method *as a class, adhering to an interface.* The Dynamic Methods are built as DynamicMethods, which are *classless.* Being classless methods, DynamicMethods can not use the `this` reference. Interface Methods however can. This in turn impacts how parameters to the methods are referenced in the IL stream. When implementing `Eval(Sheet sheet, int col, int row)` as an Dynamic Method, the `sheet` parameter is passed along as the first argument (`arg[0]`). When the same method is implemented as a Interface Method, the `sheet` parameter is passed along as the second argument (`arg[1]`), as the first argument (`arg[0]`), contains a reference to the `this` object. Thus the `Generate(...)` methods needs to be able to cope with arguments being passed along depending on wether an Interface og Dynamic Method are used for implementing the cell formula at runtime.

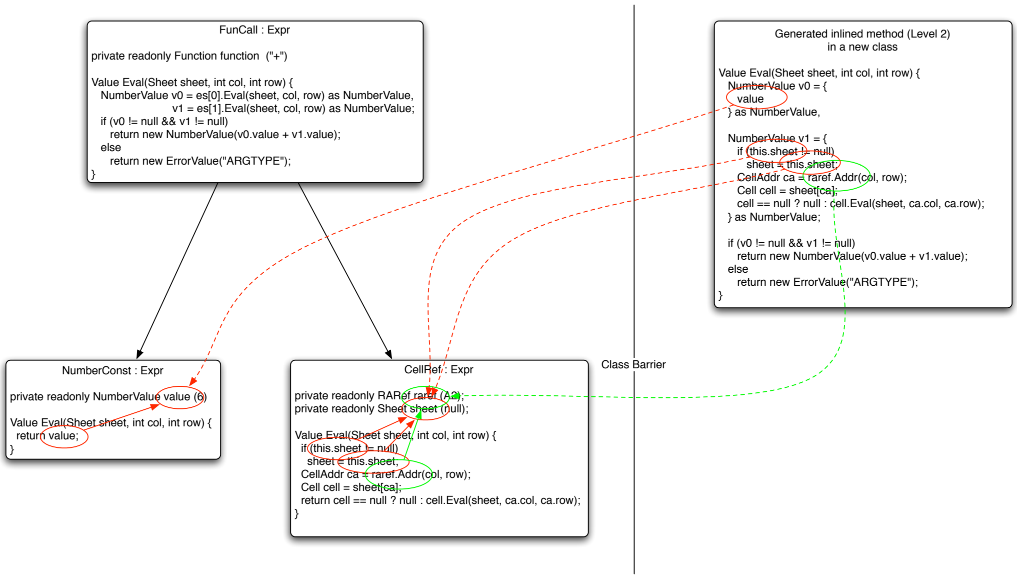
Things are in fact a bit more complicated as the methods implemented for the `Eval` methods have a different call interface than for Level 2+. For Level 1 a new method is generated for each subexpression and the generated `Eval` method expects to be called with an Array of subexpressions. This table summarizes the call interfaces and methods:

	Dynamic methods	Interface Methods
Level1	this: nonexisting Sheet sheet: `arg[0]` Expr[] expr: `arg[1]` int col: `arg[2]` ínt row: `arg[3]`	this: `arg[0]` Sheet sheet: `arg[1]` Expr[] expr: `arg[2]` int col: `arg[3]` int row: `arg[4]`
Level2+	this: nonexisting Sheet sheet: `arg[0]` int col: `arg[1]` ínt row: `arg[2]`	this: `arg[0]` Sheet sheet: `arg[1]` int col: `arg[2]` int row: `arg[3]`

The `Generator` methods uses the **GeneratorOptions** class to obtain the correct argument number for the parameters. The **GeneratorOptions** then uses its internal state about what generator method and generator level to supply the correct parameter information.

5.5.6 RTCGExprFieldInfo (parameters)

The runtime generated method for a formula need to reference data found in the (sub)expressions of that formula. As an example, consider the formula expression `A1=6+A2`. This expression consist of two subexpressions, a **CellRef** and a **NumberConst**. The **CellRef** contains two private members **RARef** and **Sheet** whereas the **NumberConst** class has a private **NumberValue**. When doing evaluation through interpretation (Level 0) each (sub)expressions `Eval` method have access to these private members which are needed for evaluation of the expression. The method generated by the generating extension also need access to these members, but being private to another class this is not directly possible. See figure 10 for an example of the problem using C# as pseudo code for the generated code.



```
private readonly Function function  ("+")

Value Eval(Sheet sheet, int col, int row) {
  NumberValue v0 = es[0].Eval(sheet, col, row) as NumberValue,
                  v1 = es[1].Eval(sheet, col, row) as NumberValue;
  if (v0 != null && v1 != null)
    return new NumberValue(v0.value + v1.value);
  else
    return new ErrorValue("ARGTYPE");
}
```

```
Value Eval(Sheet sheet, int col, int row) {
  NumberValue v0 = {
    value
  } as NumberValue,

  NumberValue v1 = {
    if (this.sheet != null)
      sheet = this.sheet;
    CellAddr ca = raref.Addr(col, row);
    Cell cell = sheet[ca];
    cell == null ? null : cell.Eval(sheet, ca.col, ca.row);
  } as NumberValue;

  if (v0 != null && v1 != null)
    return new NumberValue(v0.value + v1.value);
  else
    return new ErrorValue("ARGTYPE");
}
```

```
private readonly NumberValue value (6)

Value Eval(Sheet sheet, int col, int row) {
  return value;
}
```

```
private readonly RARef raref (A2);
private readonly Sheet sheet (null);

Value Eval(Sheet sheet, int col, int row) {
  if (this.sheet != null)
    sheet = this.sheet;
  CellAddr ca = raref.Addr(col, row);
  Cell cell = sheet[ca];
  return cell == null ? null : cell.Eval(sheet, ca.col, ca.row);
}
```

Figure 10: Generated code cannot reference private fields.

A possible solution could be to make the private members public but this is considered bad practise in object oriented programming and therefore avoided. The approach taken in TinyCalc are to let each class that derives from the **Expr** class implement a `CollectFieldInfo(List<RTCGField> list)` method which adds the private fields for that class to a list using a **RTCGField** class.

```
public class RTCGField
{
   Object obj;            // Generic object which can ref. anything.
```

```
    Expr expr;            // Expression this field is defined in
    Type valuetype;       // Type of the obejct.
                          // Object.GetType() does exist
                          // but obj _can_ be null.
    Type classtype;       // Type of class
    String fieldname;     // Name of field
    private FieldInfo fi; // FieldBuilder to be used when
                          // generating ILASM.
}
```

The list of references are then passed to the runtime generated class through the constructor for that class. While this scheme works and avoids turning private members into public ones in the expression classes, a **Object** class is utilized and this in turn make the code non type safe! As a last point it is noted that aforementioned list of **RTCGField** is converted to a hash for fast lookup of fields when generating the code.

5.6 Debugging, development aids and ILasm notes

It is useful to make debugging an integral part of the development process when using a rapid development methodology. This thesis is no exception. Developing applications that utilize native IL is at best tricky business. Often the CLR will just throw an exception stating: "Bad IL" when trying to execute a method. There is often no hint of what IL instruction caused the CLR to throw the exception.

While being a really good tool for developing code in C# or other high level languages, Visual Studio 2005 (VS2005) is not adequately equipped to help debugging IL code without leaving the IDE. That said, VS2005 comes with the `ildasm` utility which allows one to disassemble code into IL code. It is not fancy, but for what it is supposed to do, it does it rather well. However, having `ildasm` in the development loop was found to slow development down enough to be irritating, because one has to leave the VS2005 IDE, reopen and reexpand the class hierarchy tree in `ildasm` to find the changes just made to the IL code. A search for alternatives brought up three interesting methods for debugging IL:

1. It is possible to augment the generated code with *SequencePoints* and make a mapping between these sequence points and a simple textual file. Together with some magic telling the JIT compiler not to optimize the code, this enables high level debugging using the built-in debugger in VS2005. Thus it is possible make up high level statements[9] out of ILasm and singlestep these. This technique is

[9] These statements are, in fact, just strings, so they need not be C# language constructs or language constructs at all.

described in [28]. This method does *not* work on the new Lightweight Code Gen (LCG) style of emitting IL code in .NET 2.0 (DynamicMethod) but only works using the old .NET 1.1 way of generating code at runtime (Interface method).

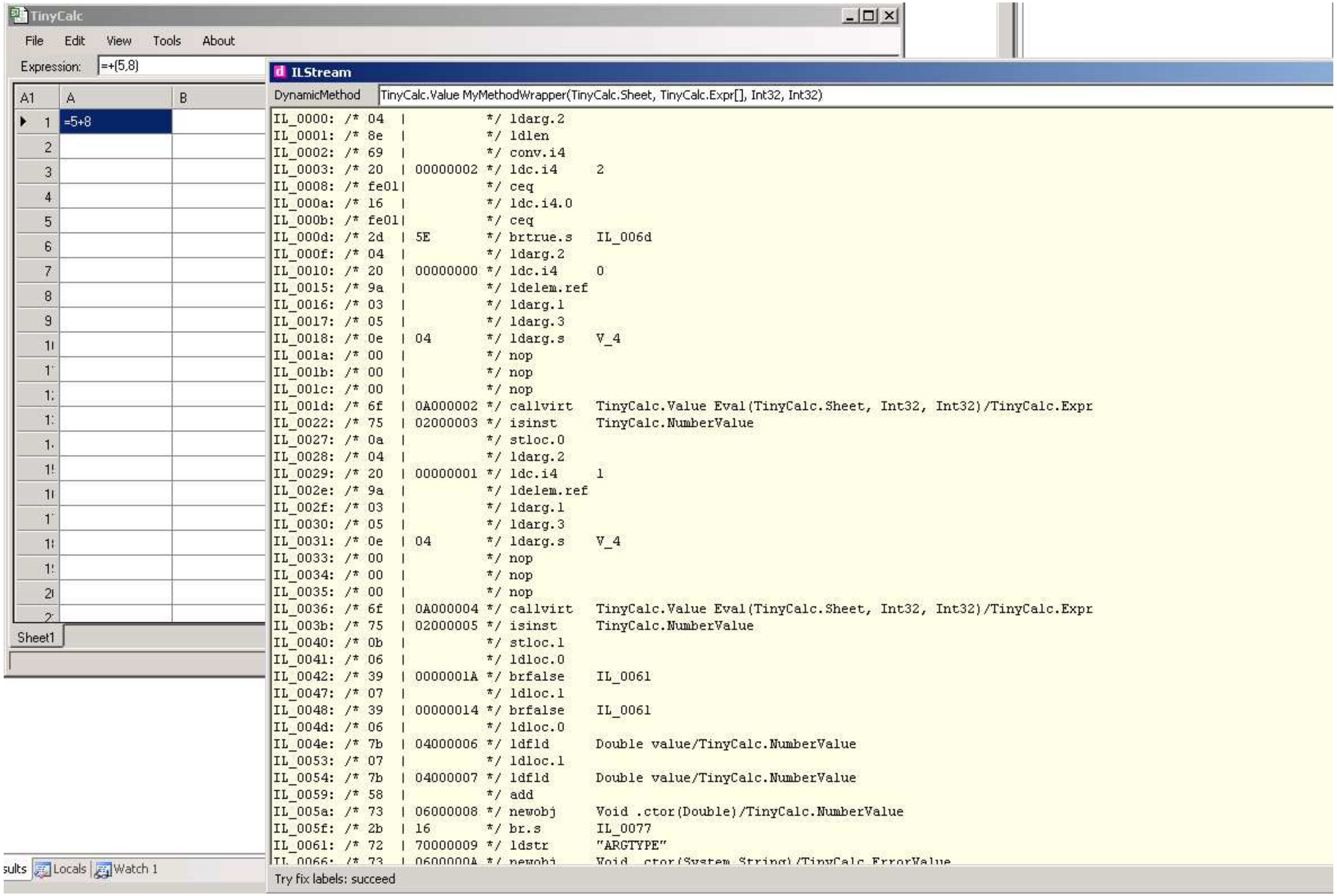
2. A visualizer for VS2005 has been written that, when given a DynamicMethod displays the ILasm of the method. A description can be found in [36]. This technique *only* works for code emitted in a DynamicMethod.
3. Mike Stall has written a debugger for managed code under .NET. This debugger is called `MDbg` [29] and is capable of debugging ILcode besides high level code. It should be noted that `MDbg` uses the *native* JIT compiled code as sequence marks for its stepsize. That is, when "single-stepping" code with `MDbg` the cursor advances in steps according to the native code, not according to the IL code.

While (1.) and (3.) are interesting and provide the ability to single step IL, single-stepping IL code is not really required in developing the generating extension in TinyCalc. Besides, (1.) requires extra code and a textual file to be written and kept in sync between the IL code and the high level description. The visualizer in (2.) is a no-frills simple solution to show IL code from within VS2005 so TinyCalc will use this to display the IL. A sample screen shot of TinyCalc and (2.) showing the IL which is about to be evaluated after the visualizer window is closed, can be seen in Figure 11.

5.7 Conclusion on augmenting TinyCalc with RTCG

TinyCalc has been augmented with RTCG and it can be concluded that it is doable to extend spreadsheet systems with RTCG. It can also be concluded that while not the easiest system to implement, it has been easier than expected. The hard part has been implementing the generating extension, notably writing and debugging the IL assembler code. This, however, only has to be done once when implementing the generating extension. The complexity of RTCG is thus well encapsulated in a small set of classes and files. Lastly it can be concluded that the generating extension need to know about private fields of expression and subexpressions. There is a tradeoff between the complexity of obtaining access to these fields and their security level. The approach taken in this thesis has been not to alter the member access modifiers as defined in the base system.

It can be concluded that a simple system like TinyCalc can be augmented with basic support for RTCG in about 5 months time by a single person. The basis implementation given by Peter Sestoft is about 1400 lines of code. Support for RTCG has added around 4500 lines of code to this number. Of those 4500 lines of code, 3800 make up `RTCG_Function.cs`. This would be a lot less if it had not been for the decision to make separate code paths for each optimization level. With GUI, command line interface, support for scripts, interpretation and RTCG, TinyCalc consists of a mere 11.000 lines of commented source code.



TinyCalc.Value MyMethodWrapper(TinyCalc.Sheet, TinyCalc.Expr[], Int32, Int32)

```
IL_0000: /* 04  |          */ ldarg.2
IL_0001: /* 8e  |          */ ldlen
IL_0002: /* 69  |          */ conv.i4
IL_0003: /* 20  | 00000002 */ ldc.i4     2
IL_0008: /* fe01|          */ ceq
IL_000a: /* 16  |          */ ldc.i4.0
IL_000b: /* fe01|          */ ceq
IL_000d: /* 2d  | 5E       */ brtrue.s   IL_006d
IL_000f: /* 04  |          */ ldarg.2
IL_0010: /* 20  | 00000000 */ ldc.i4     0
IL_0015: /* 9a  |          */ ldelem.ref
IL_0016: /* 03  |          */ ldarg.1
IL_0017: /* 05  |          */ ldarg.3
IL_0018: /* 0e  | 04       */ ldarg.s    V_4
IL_001a: /* 00  |          */ nop
IL_001b: /* 00  |          */ nop
IL_001c: /* 00  |          */ nop
IL_001d: /* 6f  | 0A000002 */ callvirt   TinyCalc.Value Eval(TinyCalc.Sheet, Int32, Int32)/TinyCalc.Expr
IL_0022: /* 75  | 02000003 */ isinst     TinyCalc.NumberValue
IL_0027: /* 0a  |          */ stloc.0
IL_0028: /* 04  |          */ ldarg.2
IL_0029: /* 20  | 00000001 */ ldc.i4     1
IL_002e: /* 9a  |          */ ldelem.ref
IL_002f: /* 03  |          */ ldarg.1
IL_0030: /* 05  |          */ ldarg.3
IL_0031: /* 0e  | 04       */ ldarg.s    V_4
IL_0033: /* 00  |          */ nop
IL_0034: /* 00  |          */ nop
IL_0035: /* 00  |          */ nop
IL_0036: /* 6f  | 0A000004 */ callvirt   TinyCalc.Value Eval(TinyCalc.Sheet, Int32, Int32)/TinyCalc.Expr
IL_003b: /* 75  | 02000005 */ isinst     TinyCalc.NumberValue
IL_0040: /* 0b  |          */ stloc.1
IL_0041: /* 06  |          */ ldloc.0
IL_0042: /* 39  | 0000001A */ brfalse    IL_0061
IL_0047: /* 07  |          */ ldloc.1
IL_0048: /* 39  | 00000014 */ brfalse    IL_0061
IL_004d: /* 06  |          */ ldloc.0
IL_004e: /* 7b  | 04000006 */ ldfld      Double value/TinyCalc.NumberValue
IL_0053: /* 07  |          */ ldloc.1
IL_0054: /* 7b  | 04000007 */ ldfld      Double value/TinyCalc.NumberValue
IL_0059: /* 58  |          */ add
IL_005a: /* 73  | 06000008 */ newobj     Void .ctor(Double)/TinyCalc.NumberValue
IL_005f: /* 2b  | 16       */ br.s       IL_0077
IL_0061: /* 72  | 70000009 */ ldstr      "ARGTYPE"
IL_0066: /* 73  | 0600000A */ newobj     Void .ctor(System.String)/TinyCalc.ErrorValue
```

Try fix labels: succeed

Figure 11: Screenshot of the Visualizer showing a sequence of ILAsm code.

6 Tests

The emphasis of this thesis is exploring new ideas with RTCG in spreadsheets, so a rigorous test is not constructed. That said, testing is still vital since a well crafted test suite will speed up the development by ensuring that source modifications and experiments do not introduce any (significant) bugs. These tests are known as regression tests.

In essence, TinyCalc consists of a little spreadsheet engine with RTCG and a set of supporting classes providing I/O methods, a GUI, a command line interface, a script interface and methods dealing with debugging and development aids scattered throughout the files. Due to time restrictions only the core spreadsheet calculation engine is subject to regression tests. The core spreadsheet engine consists of 4 parts:

1. A set of parser/scanner classes which turn text strings into formulas and constants in an internal format.
2. A set of methods and classes for (in a broad sense) initializing and setting up spreadsheets.
3. A set of methods and classes dealing with *evaluation* of spreadsheets to produce values.
4. A set of methods and classes dealing with *run-time code generation* of a spreadsheet followed by evaluation of the spreadsheet using the generated code.

The methods in (3) is largely untouched compared to the initial code delivered by Peter Sestoft and as such is considered being the basis to which the code in (4) it is being compared. The grammar and code in (1) have been augmented (section 4.2.7) compared to the grammar and code given by Peter Sestoft and only these additions are tested as it is assumed that the grammar in the basis system is correct. Lastly the I/O methods er tested by loading and saving the test sheets defined in section 4.6.

6.1 Testing the evaluation/RTCG engine

Testing the evaluation/RTCG engine is done by ensuring that the engine produces correct results at all generator levels. The test can be further subdivided into these subtests:

- At all levels ensuring that TinyCalc evaluates expressions according to the grammar rules for spreadsheets as defined in section 4.2.7. This is done by utilizing a small set of formulas exercising the extensions to the grammar.

- Perform tests that ensure that TinyCalc produces identical results at all optimization levels. This implies tests that ensures the correct and expected numerical value is returned at all levels, as well as tests ensuring that errors are caught at all levels. Testing for errors can be further subdivided into two tests:
    1. Errors resulting from an operator or function being applied to a wrong number- or type of arguments. This type of errors result in an error *value* instead of a number, string or matrix.
    2. Errors resulting from fatal exceptions, ie. when a formula tries to reference a cell outside of the sheet and so forth.
- When using formula sharing, ensure that the semantics for formula sharing can handle that the shared formula might not produce the identical kind of result as described in 5.4.
- A set of adhoc tests which are found to be useful during development.

6.2 Testing the additions to the grammar

Testing the additions to the grammar consist of testing that:

- The power operator is parsed with the right precedence.
- Scientific numbers are parsed correctly.
- References in R1C1 style are parsed correctly.
- The string concatenation operator (&) can be parsed and has the correct precedence.
- That sheet references can be parsed.

6.3 Testing the I/O methods

Testing the I/O methods constitute of

- loading the XMLSS test sheet checking that it could be loaded correctly.
- loading GNUMERIC test sheet checking that it could be loaded correctly.
- saving one of the test sheets above in XMLSS format.
- loading the newly saved XMLSS sheet and checking that it could be loaded correctly.

If these 4 points can be performed without errors the I/O methods are assumed to be working.

6.4 Test conclusion

The tests has been implemented and an example of these tests can be seen in section A.7. The full set of tests can be located in the `Regression-Tests` directory on the CD-ROM. To run the test, issue an `runtests.bat` in that catalog. TinyScript (section 4.5 is utilized for testing which makes it possible to write a test in a single line. See `expression-tests.txt` as an example. A total of approximately 350 regression tests are conducted.

The tests has been a success [31] in the sense that errors *were* found and corrected. That said, no major errors were found by the test procedure. The errors found are listed below along with their status:

- The statement `ilg.ThrowException(typeof(ArgumentException));` causes an error when used in a DynamicMethod:

  > Unable to cast object of type 'System.Reflection.Module' to type 'System.Reflection.Emit.ModuleBuilder'.

  The statement appears in the `ApplyAct` methods and can be exercised by trying to compute the formula `A1=SUM(5,"Thomas")` using the Dynamic Method call interface. The problem is *not* resolved for the time being.

- The grammar extension supporting sheet references using the '!' character as separator was found to clash with the "not equal" token '!=' in the grammar. An example showing this problem is the formula `A1=A2!=A3`. The parser would parse `A2!` as a sheet reference to a sheet named `A2` and complain that `=A3` was not a valid cell reference. To remedy this situation the "not equal" *has been* converted to '<>' in the grammar. This is consistent with Excel, OOCalc and Gnumeric.

- The invariance rules for the abstract stack machine was found not to obeyed in a single spot in the generator. This caused an exception during runtime generation. The mistake has been corrected.

- The stack invariance rules for the CLR was not obeyed for variadic functions. These rules in essence states that no matter how the flow through the code might be at runtime, the CLR evaluation stack need to be compile time deterministic and consistent for the JIT compiler. `SUM(A1:A5)` did not exhibit the problem but `SUM(SUM(A1:A5)` did as the inner `SUM` function could potentially cast an exception causing objects to remain on the evaluation stack that would not be there if the computation had progressed without error. The error *has been* corrected.

- The I/O tests showed that Gnumeric and Excel encodes single cell matrix values differently. Gnumeric treats them as matrix values, whereas Excel encodes them as number values. Nothing has been done about this as it is unclear how single cell matrix values are to be interpreted in CoreCalc.
- It was found that `SUM(A1:A3)` worked but `SUM(A3:A1)` did not. As this problem also was present in CoreCalc, Peter Sestoft was informed and he produced a patch which has been incorporated into TinyCalc. Furthermore the generating extension *has been* augmented with a corresponding generator method.
- Lastly it was found that only exceptions of type "Cyclic" resets the computation flags to a valid and known state. If for instance an formula causes an "IndexOutOfBounds" exception this exception is not caught by the calculation logic and the computation flags are not reset and some cells will figure as being uptodate others needing recalculation. Any subsequent recalculation will (probably) produce invalid results. The problem *has been* resolved by catching *all* exceptions.

With these tests it is concluded that it is very unlikely that TinyCalc contains any major bugs in the code generator as the generated code in all 350 tests produces identical and correct results to the results obtained by interpretation.

7 Performance Evaluation

A couple of small benchmarks were performed on TinyCalc. These benchmarks were performed to gain information in two areas:

- What impact does RTCG have on the calculation time in TinyCalc? What speedup can be obtained at the different optimization levels?
- How does TinyCalc fare against full featured spreadsheets, most notably Microsoft Excel, OOCalc and Gnumeric?

The benchmarks were constructed so that recalculation took a considerable amount of time, whilst still only used features implemented in TinyCalc, that is simple arithmetic operations on doubles, cell reference, cell areas and simple function calls. Keeping the benchmarks simple also makes it easier to understand whether a given optimization is good or bad for the recalculation time. Each "simple" primitive formula might not take a long time, but to remedy that, these simple formulas were duplicated many times. This benefitted the benchmark in two ways: First of all, it made the recalculation take enough time for it to be measured reliably. Secondly, using duplicated formulas made the time used for compilation of formula expressions negligible, as formula sharing was possible. This allowed us to concentrate on the performance gain of the generated code instead of the overhead. Three sets of benchmarks were constructed:

1. Many duplicated formulas performing a lot of arithmetic. With and without cell references. A Taylor series expansion of $\exp(x)$ was calculated.
2. Simple calls to a spreadsheet function which is translated to a native C# function call, eg. sin(0.5) and sin(A1).
3. Long reference chains using literally millions of cell references.

7.1 Benchmark setup

All benchmarks are performed on a laptop with an Intel Pentium-M Dorthan CPU with 2MB Level 2 cache, 768MB ram. The Pentium-M CPU is equipped with Speedstep enabling it to run at various speeds. While conducting the benchmarks the cpu are kept at 600MHz. The system is kept as idle as possible during the benchmarks: all nonessential programs are closed and the system is left alone without user intervention. On the software side, the setup is:

- Operating System: Windows 2000 professional (Microsoft Windows NT 5.0.2195 Service Pack 4)

- Excel 2003 (Version 11.5612.5606)
- OOCalc (Version 2.0.2)
- Gnumeric for Windows version 1.6.3
- Visual Studio 2005 (8.0.50727.42)
- Microsoft .NET Framework Version 2.0.50727

TinyCalc uses a Stopwatch class (new in .NET 2.0) to measure time. Its precision depends on the hardware on which the code is running. If the hardware supports it, it uses a high resolution hardware performance counter; if it does not, the system timer is used instead.

Example 12 Example of using Stopwatch class to time execution time

```
Stopwatch watch = new Stopwatch();

// How precise are the measurements?
long frequency = Stopwatch.Frequency;

watch.Reset();
watch.Start();
... long running calculation ...
watch.Stop();
watch.ElapsedMilliseconds;
```

Recalculation time in Excel is measured using a Visual Basic macro (invoked by pressing `ALT-F8`). The resolution of the Timer in Excel/VB is not known:

Example 13 Macro for timing recalculation in Excel

```
Sub Recalculate()
   ' Recalculate Macro
   ' Macro recorded 27-05-2006 by Thomas S. Iversen
   timing = Timer     'Floating point register used
      Application.CalculateFullRebuild
   timing = Timer - timing
   timing1 = Timer
      Application.CalculateFull
   timing1 = Timer - timing1
   MsgBox "CalculateFullRebuild: " & Format(timing,"0.000") & " seconds"
       & vbCrLf
       & "CalculateFull: " & Format(timing1, "0.000") & " seconds"
End Sub
```

The difference between `CalculateFull` and `CalculateFullRebuild` is that the latter rebuilds the dependency graph(s) for the workbook before performing a full recalculation. It is included due to the fact that some people [24], [7], [7], find it necessary to use this recalculation method in order for the recalculation to work. It is unclear why this is so, but it seems related to large workbooks with complex dependencies and/or user defined functions.

Recalculation time in Gnumeric is measured using a hand held stopwatch. While Gnumeric has an Python based plugin API, it does not work under Windows, so the old-fashioned method has to be used.

The macro used for performing recalculations in OOCalc is identical with the one used for recalculations in Excel, except that in OOCalc the recalculation method is named `ThisComponent.calculateall()` instead of `Application.CalculateFull`.

Screenshots taken of the benchmarks performed in Excel and OOCalc makes it easier to remember the actual numbers. TinyCalc implements the benchmarks in TinyScript (Section 4.5), but as this still requires tedious work in order to make charts from the raw data, a helper class called **TinyBench** has been implemented. This class can produce a XML representation of the benchmark data. A utility `TinyBench2Ploticus` has been implemented. This utility takes XML data (including third party data for Excel, OOCalc and Gnumeric), a Ploticus[15] plot template and then generates a ploticus plotscript. After this initial programming, the benchmarks and constructions of charts can be fully automated. See section A.8 in the appendix for further information on how to use TinyBench. Each benchmark in TinyCalc is run 3 to 5 times and the average runtime computed. All benchmarks and their results can be located on the CD-ROM in the `TinyCalc/Benchmarks/Script` directory and its subdirectories.

7.2 Taylor benchmarks

The basic idea in the Taylor benchmarks is to use a lot of simple operations such as multiply, division, addition and combine them with cell references to get an expression that is simple and yet still take time to calculate.

A Taylor series expansion of exp (0.5) can be expressed as:

$$\exp(0.5) = 1 + \frac{0.5}{1!} + \frac{0.5^2}{2!} + \frac{0.5^3}{3!} + \ldots + \frac{0.5^n}{n!}$$

While the pow (ˆ) operator *is* implemented in TinyCalc, the purpose is to use as many simple operations in these benchmarks, so the pow terms is replaced by simple multiplications. The factorial function (!) is also expanded into a series of multiplications:

$$\exp\left(0.5\right) = 1 + \frac{0.5}{1} + \frac{0.5 * 0.5}{2 * 1} + \frac{0.5 * 0.5 * 0.5}{3 * 2 * 1} + \ldots + \frac{0.5 * \ldots * 0.5}{n * (n-1) * \ldots * 1} \qquad (15)$$

As Excel can handle up to 1024 chars in an expression, n is set to 13. 131072 copies are then made of this formula (32 columns, 4096 rows) and the time taken to recalculate the whole workbook is measured.

7.2.1 Taylor benchmark — no references

This benchmark uses formula (15) as presented in section 7.2. The benchmark and corresponding result can be found in the subdirectory `TaylorNoReferences`. Figure 12 shows the results.

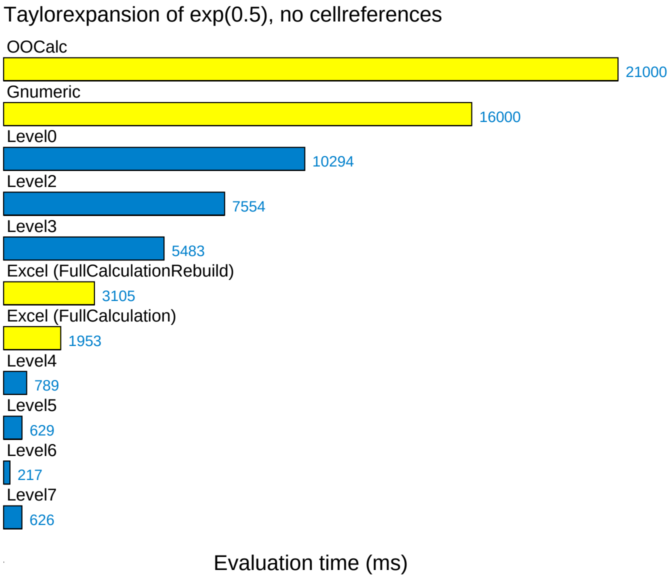


Figure 12: Taylor benchmarks — no references

It is interesting to note that using interpretation (Level 0) TinyCalc outperforms both OOCalc and Gnumeric. It can be seen that inlining all the subexpressions gives a performance speedup (Level 2). Removal of type checks (Level 3) is also worthwhile. Level 4 shows that avoiding construction of temporary **NumberValues** in long computation sequences is very beneficial. Inlining of constant values (Level

5) gives a little speedup and Level 6, where the double values are kept directly on the stack, brings the time down to an absolute minimum. Remembering that Level 7 equals Level 5 with speculative external type deduction, the times for Level 7 are to be expected as there are no external cell references in this example.

7.2.2 Taylor benchmark — Argument is referenced

Formula (15) is not very useful as the argument is hard coded into each formula. In this benchmark the formulas are modified so that they reference an argument in A1:

$$\exp\left(\mathrm{A1}\right) = 1 + \frac{\mathrm{A1}}{1} + \frac{\mathrm{A1} * \mathrm{A1}}{2 * 1} + \frac{\mathrm{A1} * \mathrm{A1} * \mathrm{A1}}{3 * 2 * 1} + \ldots + \frac{\mathrm{A1} * \ldots * \mathrm{A1}}{n * (n-1) * \ldots * 1}$$

where $\mathrm{A1} = 0.5$. The benchmark and corresponding result can be found in the subdirectory `TaylorReferenceArgument`. Figure 13 shows the result as a chart.

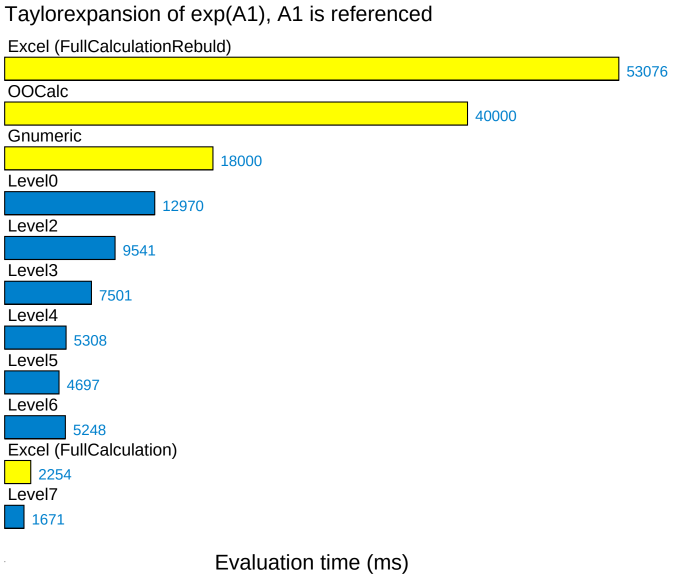


Figure 13: Taylor benchmarks — Argument is referenced

TinyCalc is again faster than OOCalc and Gnumeric at Level 0. Inlining and type deduction helps (Level 2 ... 6), but it is only at Level 7 that TinyCalc can deduce that A1 indeed is a **NumberValue**. When that happens, however, type checks can be removed and TinyCalc ends up being faster than Excel. Notice, also, how Level 6 is slower than Level 5 when the double values cannot remain on the CLR stack between calculations. This need not be the case and implementing a special case in the generator would make Level 6 and Level 5 equally fast in that circumstance. As it is a minor problem, this is postponed.

7.2.3 Taylor benchmark — All references

In this tests the denominators of the formula expressions are also converted to cell references:

$$\exp\left(\mathrm{A1}\right) = 1 + \frac{\mathrm{A1}}{\mathrm{B1}} + \frac{\mathrm{A1} * \mathrm{A1}}{\mathrm{B2} * \mathrm{B1}} + \frac{\mathrm{A1} * \mathrm{A1} * \mathrm{A1}}{\mathrm{B3} * \mathrm{B2} * \mathrm{B1}} + \ldots + \frac{\mathrm{A1} * \ldots * \mathrm{A1}}{\mathrm{B}n * \mathrm{B}(n-1) * \ldots * \mathrm{B1}}$$

Where $\mathrm{A1} = 0.5$ and $\mathrm{B}n = n$. The benchmark and corresponding result can be found in the subdirectory `TaylorAllReferences`. Figure 14 shows the result as a chart.

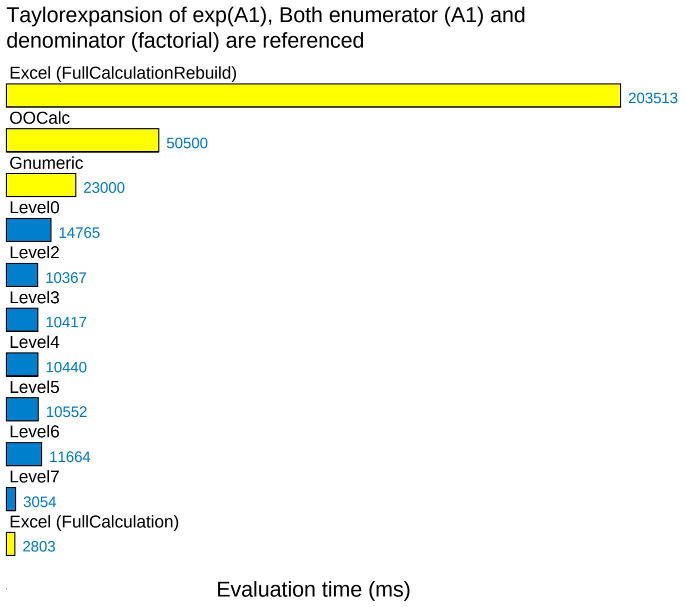


Figure 14: Taylor benchmarks — All references

Again is it possible for TinyCalc to outperform OOCalc and Gnumeric at Level 0. Level 2 provides a little speedup as subexpressions are inlined. Again, it is only at Level 7, where it can be deduced that $\mathrm{A1}, \mathrm{B1} \ldots \mathrm{B13}$ are **NumberValues**, that TinyCalc can perform the recalculation nearly as fast as Excel. It is also evident that

Excel has some problems with a FullCalculationRebuild when a workbook contains lots of references.

7.2.4 Taylor benchmark — All references optimized

In order to minimize the number of calculations performed over and over again, the benchmark is reexpressed:

$$\exp\left(\mathrm{A1}\right) = 1 + \frac{\mathrm{A1}}{\mathrm{B1}} + \frac{\mathrm{A2}}{\mathrm{B2}} + \frac{\mathrm{A3}}{\mathrm{B3}} + \ldots + \frac{\mathrm{A}n}{\mathrm{B}n}$$

Where $\mathrm{A}n = 0.5^n$ and $\mathrm{B}n = n!$. The benchmark and corresponding result can be found in the subdirectory `TaylorAllReferencesOptimized`. Figure 15 shows the result as a chart.

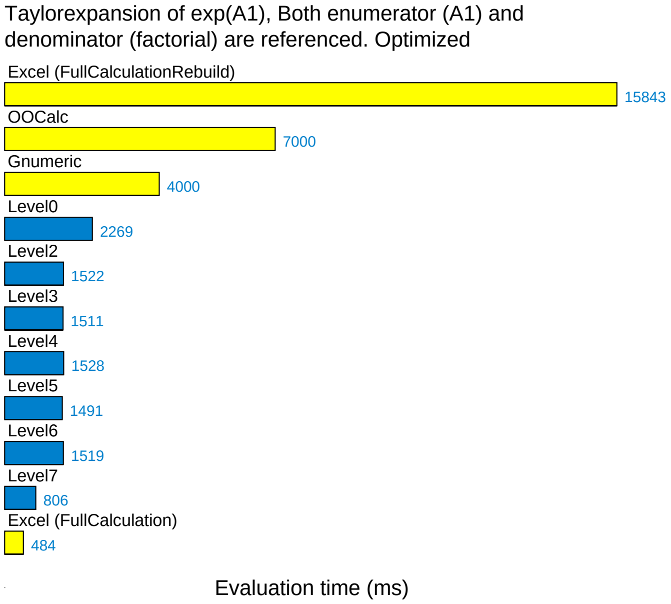


Figure 15: Taylor benchmarks — All references optimized

TinyCalc is again faster than OOCalc and Gnumeric. Inlining helps a little, but only external type deduction can speed the calculation up any further. Even so, Excel beats TinyCalc by approximately a factor 2.

7.3 Simple Math function

In this benchmark, a simple function call will be benchmarked. These calculations performs almost no calculation inside the spreadsheet calculation engine. Instead, a (heavily) optimized method is called in an external library. The first benchmark performs 262144 calculations of $\sin(\pi/4)$. The TinyScript script and corresponding output can be found in the subdirectory `SimpleStaticMathFunction` on the CD-ROM and figure 16 shows the numbers in a chart.

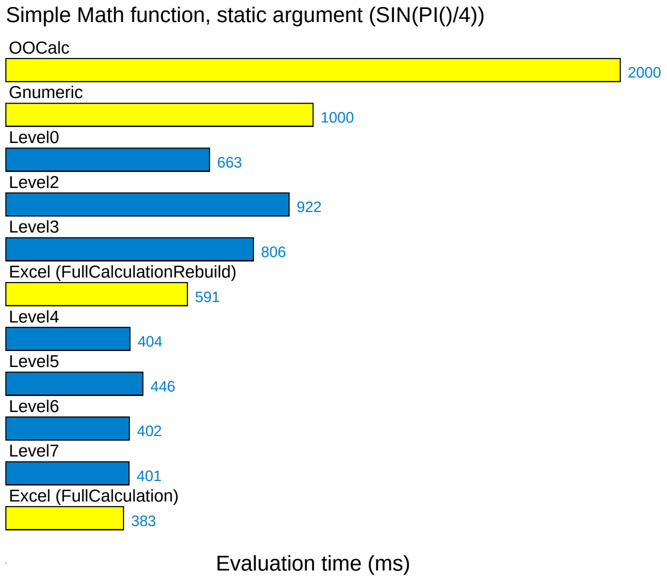


Figure 16: Simple Math function, SIN(PI()/4)

It can be seen that all 3 spreadsheets can perform the calculations almost equally fast. The benchmark is performed so fast that external events (disk access and the like) can skew the results quite considerably, and as such care should be taken concluding *too* much from these benchmarks. It should, however, be noted, that it appears that avoiding temporary objects can be beneficial. It is only one object per calculation, but as the benchmark consists of 262144, calculations it amounts to 262144 calls to the memory management code in C#, which needs to synchronize with the garbage collector. Level 2 and Level 3 are most likely slower than Level 0 due to overhead by generating and invoking the code.

The next benchmark consists of 262144 calculations of $\sin(A1)$, where $A1 = PI()/4$. The TinyScript script and corresponding output can be found in the subdirectory `SimpleReferenceMathFunction` on the CD-ROM and figure 17 shows the numbers in a chart.

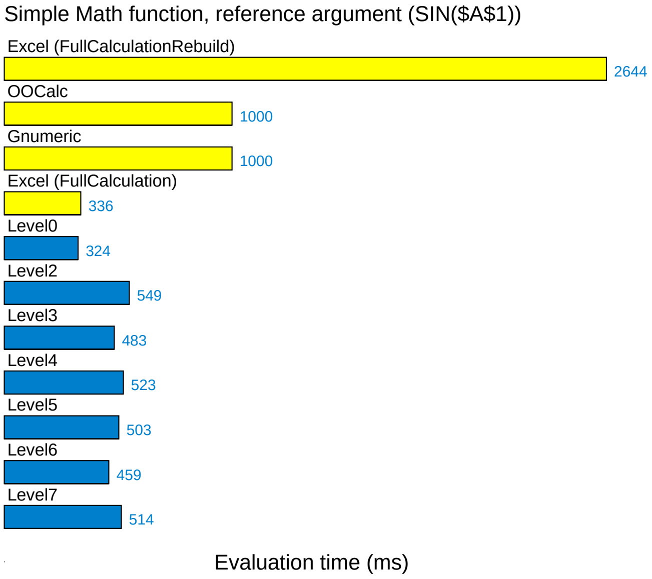


Figure 17: Simple Math function, SIN(A1)

Notice how Excel and TinyCalc are equally fast at Level 0. TinyCalc performs the benchmark approximately equally fast at Level 2 through 7. This is most likely because of overhead and the fact that there are no temporary objects that TinyCalc can avoid creating as there was in the first simple benchmark.

The one and only conclusion drawn from this benchmark is that, for simple functions which utilizes an external library TinyCalc is as fast as Excel, OOCalc and Gnumeric, and that the overhead of doing RTCG on these simple calls might not be beneficial. On the contrary, it proved to slow the calculations down this benchmark.

7.4 Long reference chains

The two previous benchmarks had limited cell reference utilization. To benchmark how TinyCalc, Excel, OOCalc and Gnumeric cope with literally millions of references, this benchmark was constructed. Again, it is quite simple. In the `A` column, a series of simple interest formulas are calculated. More specifically, A1 = number, A2 . . . A12288 = R[−1]C ∗ 1,00001. In the `B` column, 12288 partial sums are constructed. More precisely B1 = SUM(A$1 : A1) and this formula is then copied to

B2…B12288 yielding a total of 12288 references from the A column and approximately $(12288 + 1)/2 * 12288$ cell references from the B column. In total, over 75 million cell references using only two distinct formulas when using formula sharing!

Due to time constraints, this thesis has not considered RTCG for **MatrixValues** in general, but specially for this benchmark, the **CellArea** class and `Eval` and `Apply` methods were implemented so that code for `SUM` could be generated. Performance for Level 2 through 7 are expected to be equal to the performance of Level 0 as almost no optimization has been performed.

The TinyScript script and corresponding output can be found in the subdirectory `LongReferenceChains` on the CD-ROM. Figure 18 contains the results as a chart.

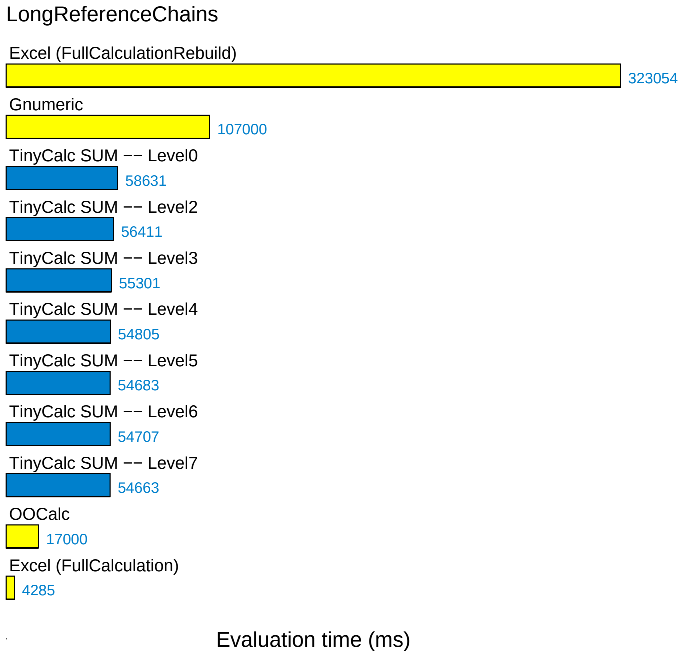


Figure 18: Long reference chains

As it was expected, performance of Level 0 (interpretation) is the same as Level 2 through 7. TinyCalc is faster than Gnumeric (by a comfortable margin), but is a lot slower than Excel. Excel is very fast and uses about 32 cyles[10] per addition and reference pair which is impressive.

The differences in recalculation time between Excel and TinyCalc is so big that it was investigated why this was so. Performing crude optimizations on the IL code

[10] 75 million addition and references in 4 seconds at 600Mhz

emitted for **MatrixValue** calculations proved worthless. Eventually, it was found that it boiled down to the way **MatrixValue** is implemented in TinyCalc. The **MatrixValue** class uses an **Value[,]** array to hold the actual values a specific **MatrixValue** contains. That is, evaluating `SUM(A1:D4)` evaluates the cell area `A1:D4`, which in turn returns a **MatrixValue** in which a **Value[4,4]** array has been allocated and initialized with references to the 16 underlying values. It is the assignment of these 75 million references that slow TinyCalc down when performing the benchmark above.

It is fair to say that the CoreCalc (which TinyCalc has inherited) representation of **MatrixValues** is overly general using a **Value[,]**. In contrast to Excel, it allows matrices containing a mixture of numbers and matrices, possibly containing further matrices and so forth. In TinyCalc **MatrixValues** can for instance model general binary trees something Excel can not. Two alternative **MatrixValue** implementations are proposed:

1. A class **LWMatrixValue** which contains a tuple `(sheet, ulCa, lrCa)` defining the sheet, the upper left cell and the lower right cell defining this matrix. In essence, this defines a matrix in terms of a *view* or *cursor* on the underlying sheet. This has the implication that these matrix values only works when a function that uses them would expect a 1:1 mapping between the values as they appear in the sheet and as they appear in the matrix value. `SUM(A1:B4)` would work with matrices implemented by **LWMatrixValue** as there exists a 1:1 mapping between the the **LWMatrixValue**induced by `A1:B4` and the sheet. `SUM(MMULT(A1:B4;C1:D4))`, however, would *not* work with matrices implemented through **LWMatrixValue**. The speed of **LWMatrixValue** is expected to be good.
2. A class **DoubleMatrixValue** which implements matrices using a `double[,]` instead of a `Value[,]`. Both the direct and indirect usage of matrix values, as seen in the previous point, will work with a **DoubleMatrixValue**. There are subtleties with **DoubleMatrixValue** which does not exist in **MatrixValue**. What about null cells? How are they represented using a `double[,]`? Or how are matrices where not all cells are numbers handled? These semantic problems are deferred in this thesis, but should be considered later on. The performance of **DoubleMatrixValue** is expected to lie in between **LWMatrixValue** and **MatrixValue**.

Using the two definitions above, two functions `FASTSUM` and `DOUBLESUM` are implemented. They compute the sum as `SUM` does, but utilize **LWMatrixValue** and

DoubleMatrixValue respectively and have the limitations outlined above. Figure 19 shows that `FASTSUM` gives a remarkable speedup and `DOUBLESUM` a decent speedup at Level 0.

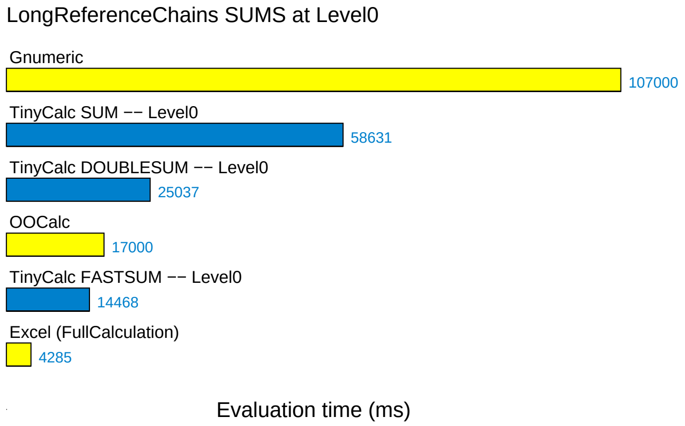


Figure 19: The three implementations of SUM at Level0

As both methods look promising, the generating extension is augmented with methods that can generate code for both of them. The optimizations possible for either of the two methods differs *and* does not follow the level scheme used for optimizations on expressions (section 5.2). A consequence of this is the way optimizations is implemented as levels needs to be rethought and reimplemented using another scheme. Suggestions on how to remedy the situation in the future are presented in section 8.3. For now, we *redefine* the optimizations levels for these two functions:

Level	FASTSUM	DOUBLESUM
Level 0	Interpreted in C#	Interpreted in C#
Level 2	As IL code. Evaluation of arguments are inlined.	As IL code.
Level 3	As Level 2 and also inlining and generation of the `(v as LWMatrixValue).Apply(act)` call so that the innermost loops can be optimized. First innermost loop optimization which is done at Level 3 is to inline the 75 million delegate calls.	Inlining of the delgate
Level 4	As Level 3 but optimize loop overhead by moving offset calculations outside of the loops. This optimization is also possible in C# code.	N/A.
Level 5	As Level 4 but assume that speculative type checking is working and that if a cell is non-null so will the resulting value be of **NumberValue** type.	N/A.

With these optimizations, two new benchmarks were run. The benchmark script and corresponding output can be found in the subdirectory `LongReferenceChainsFastSum` and `LongReferenceChainsDoubleSum` on the CD-ROM. Figure 20 contains the results as a chart.

As can be seen, very little speedup can be obtained by utilizing RTCG on **DoubleMatrixValue**. A further investigation showed that the overhead of allocating the array is 4 seconds. Traversal of the array when populating it with a takes 6 seconds. The actual additions takes about 2 seconds and the traversal fetching the values for the sum takes another 6 seconds. The last 3 seconds overhead not belonging to a specific part of the code. It is not possible to shave much time off **LWMatrixValue** either. The computation of the actual sum takes 2 seconds as for `DOUBLESUM` and 6 seconds are used to reference the actual values giving a total of 8 seconds. This is

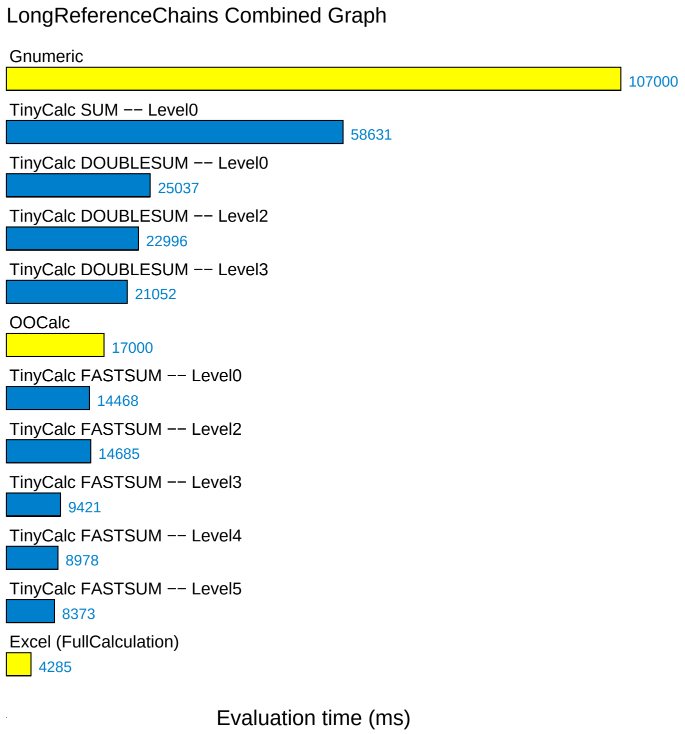


Figure 20: Long reference chains combined graph

as fast as these implementations of **MatrixValue**s get. Besides that the **LWMatrixValue** has the limitation of being a view as discussed above, but it is speculated that most applications of `SUM` on a matrix is actually of the kind where the matrix has a 1:1 mapping to the underlying sheet. As such **LWMatrixValue** has its place in TinyCalc.

7.5 Performance conclusion

It can be concluded that TinyCalc *without* RTCG is faster than OOCalc and Gnumeric in almost all cases. The exception being that **MatrixValue** is to general making `SUM` slower than it ought to be. Implementing `FASTSUM` and `DOUBLESUM` shows that TinyCalc at Level 0 can compute sums as fast as OOCalc and Gnumeric.

With RTCG TinyCalc approaches or outperforms Excel in the benchmarks. When using TinyCalc as a calculation engine and the calculation can be optimized to be performed on the CLR stack TinyCalc shines compared to Excel. Calculation of formulas containing a lot of cell references is slower in TinyCalc than in Excel.

The main conclusion is that RTCG speeds up TinyCalc by a factor of two to four times on average and much more in some benchmarks, most notably those utilizing TinyCalc as a calculation engine.

8 Evaluation

8.1 Evaluation of the process

Extending TinyCalc with a GUI, I/O methods and RTCG has been easier than expected even though it has been hard at times. That said, hours have been used searching for documentation on Excel, OOCalc, and Gnumeric and for further reference the collected information has been documented in this thesis, whenever possible.

As it is also often the case, this thesis has raised more questions than it has answered. A section on (some) of the interesting directions one could go with TinyCalc is presented.

Obtaining documentation on how Excel works and documentation on how Gnumeric is *meant* to work has been troublesome. Microsoft clearly can not disclose the source code for Excel, but a small paper describing the overall recalculation strategy along with the major data structures would have been more than enough. Similarly very little documentation on how Gnumeric and OpenOffice Calc are supposed to work can be obtained. In these latter examples the source code can be obtained, but a small paper would be much easier to read and understand than digging the information out of several thousands lines of code.

As a reference for future work, the documentation found will be listed here:

- Microsoft Developer Network has an article [23] about how formulas are recalculated in Excel 2002 (and previous versions). While being ambiguous and repetitive and raises further questions, it does provide valuable information, especially regarding why user defined functions in Excel can be evaluated multiple times with incomplete parameters during a global recalculation. The article is, however, not enough to deduce the exact recalculation algorithm for Excel.

- A company called Decision Models is specialized in speeding up Excel worksheets. More importantly, they have published [30] all the information they have gathered about how Excel actually works.

- A blog entry [8] by a Microsoft Excel developer lists what appears to be an exhaustive list of limits in the upcoming Excel2007 as well as the limits imposed in previous versions of Excel. The big question arising when reading the list is: "why do these limits exist in Excel; even Excel2007 is limited". The list might be used to deduce how Excel is implemented.

- Two MSDN articles [27] and [26] describes the XMLSS format used as XML format by Excel2002. The articles are good, but lack information about the grammar used for formulas.

- Various attempts has been made to construct a BNF grammar for Excel formulas, most notably [1] and [4].
- The difference between `CalculateFull` and `CalculateFullRebuild` is clear enough. The former recalculates the entire workbook, and the latter rebuilds the dependency structure before doing a full rebuild. What is interesting, however, is that some people ([24], [7], [7]) have trouble performing a proper recalculation using the former method when they have not changed the dependency structure in a workbook but only changed values. It is unclear in what circumstances it is safe to use the faster `CalculateFull` over the much slower `CalculateFullRebuild`.

8.2 Prior art

Has RTCG been performed in spreadsheets before? No technical article exists on the subject but two patents, US Patents 5471612[5] and 5633998[6] are titled respectively:

> Electronic spreadsheet system and methods for compiling a formula stored in a spreadsheet into native machine code for execution by a floating-point unit upon spreadsheet recalculation.

and

> Methods for compiling formulas stored in an electronic spreadsheet system.

These patents were issued by Borland in 1995 and 1997 and both details a method for performing runtime code generating of spreadsheet formulas. Compared to TinyCalc the described method compile formulas to Intel x87 FPU code as opposed to CLR byte code and contains a different recalculation model than TinyCalc. While the patent and TinyCalc both utilize runtime code generation to speed up spreadsheet calculations and have comparable ideas for some kind of abstraction level the implementations is different. It is unclear who possesses these patents as of writing (July 2006). It is also unclear if the methods described in the patents ever has been implemented in an actual spreadsheet application.

Besides these patents no prior examples, patents or articles describing how to pair runtime code generation with the spreadsheet recalculation model could be found.

8.3 Where to go from here

As it is often the case, investigation of new ideas and concepts spawns a lot of new work and ideas. Below is a list of things which could be investigated in later projects. The list can be divided into two sections, one consisting of extending TinyCalc as a calculation engine, that is, the datatypes, valuetypes, recalculation model, RTCG and so forth. The other part concerns the I/O (in a broad sense) of spreadsheets. Many of these projects can easily be generalized and need not to be restricted to the actual implementation of TinyCalc.

- Separating the command line interface code and GUI code from the rest of the project, that is turn TinyCalc into a DLL from which a stand alone command line version and a stand alone GUI version could be built. Estimated time frame is one month.
- Implementing GUI for loading, editing and saving TinyScript scripts. Estimated time: A couple of weeks.
- Reconsider what place **DoubleMatrixValue**, **LWMatrixValue** and the original **MatrixValue** have in TinyCalc. Should they coexist? Or should one be selected and the two others scraped? The semantic consequences of **DoubleMatrixValue** and **LWMatrixValue** should be analyzed. Time frame: 1-2 months.
- Extending RTCG support for matrices when the **MatrixValue** classes are in place. The generator supports **DoubleMatrixValue** and **LWMatrixValue** as of now, but further optimizations should be considered. Time frame: 1-2 months.
- Investigate the frequency with which spreadsheet functions are used and how their running time affect the total recalculation time of spreadsheets. Which functions are the most "problematic"? Which functions is the best candidates for optimization? Estimated time frame: 2-3 months as it takes time to collect problematic spreadsheets.
- Adding support for more spreadsheet file formats, most notably read support for ODF. Estimated time frame: max 1 month.
- Investigate if anything can be done about the problem of calling RTG code efficiently. Section 5.5.2 clearly showed that the Interface method of executing RTG code has problems with the asymptotic run time, as the number of generated functions induce more and more classes to be instantiated into the CLR.

Investigate whether it is possible to prune unused runtime generated code from memory? Or can the DynamicMethod be made faster? Estimated time frame: 1-1.5 month.

- Implemented the concept of named cells and named ranges. A company called Decision Models (`www.decisionmodels.com`) makes a living by selling a program called `fastexcel` that can take an arbitrary Excel workbook and produce an identical workbook in which named cells and named ranges has been avoided for speed reasons. This suggests that named cells and named ranges might be harder than one thinks. Estimated time frame: 1-2 months.
- Implementing code that, at runtime, analyzes what optimization is best suited for a given spreadsheet. And further, reconsider the optimization strategy implemented in TinyCalc. As it is now, Level n always do what Level $n-1$ did, and then some more, except Level 6. Maybe it would be wiser to switch to a generator model, where the optimization levels are defined according to what options are turned on and off. Different optimization strategies could then be performed on different datatypes (**NumberValues** are different beasts than **MatrixValues**) and furthermore the optimization level could follow the data instead of being global to a sheet. Knowledge of the JIT engine could also be put into this heuristic. Estimated time frame: 3-4 months depending on ambitions.
- Augment TinyCalc with the possibility of tracking types of external references. That is, implement some sort of dependency structure, so that when a change to the value in a cell cause the value to change type, any dependent formulas are recalculated. Estimated time frame: 3-5 months.
- Implement dependent recalculation, so that only affected formulas of a change to a cell are recalculated. As it is now, all formulas are recalculated. For what kind of spreadsheets is this simple strategy enough? And what situation will benefit greatly from a smarter recalculation engine? And how would that impact the simple structure of TinyCalc? Estimated time frame: 3-5 months.
- Implementing support for saving RTGC to permanent storage so that a spreadsheet can be optimized with RTCG once so that when it is loaded from storage the next time, it is ready to perform recalculation using RTG code without first compiling the code. Should the user change a couple of values before the sheet is recalculated only the affected expressions need to be recompiled, and that is only if the values changes types. This optimization would help in the case

where a large and time demanding but more or less static spreadsheet is used frequently. Estimated time frame: 3-4 months.

- Splitting up `RTCG_Function.cs` into smaller files. All spreadsheet functions in the basis implementation live in `Function.cs` and to make a 1:1 mapping a single file, `RTCG_Function.cs`, was created containing all the RTCG versions of the same functions. This decision, however, made `RTCG_Function.cs` quite large and splitting `RTCG_Function` into smaller files, preferably one for each function, would be advisable. Time did not permit this to be done as part of the thesis. Estimated time frame: less than 1 month.
- Combining other spreadsheet ideas done under supervision of Peter Sestoft, ITU, into TinyCalc. Especially the idea of making user definable functions (UDF) as spreadsheets themselves seems like a intriguing concept to combine with RTCG. Estimated time frame: 2-3 months.
- Correct the two remaining problems found in section 6.4
- In general, perform research on spreadsheet systems and spreadsheet technology. A search for technical articles and research papers on spreadsheets divides into two groups of papers. The first groups contain papers on solving various (mathematical or intensive) problems using spreadsheets. The other group contain papers on how to minimize spreadsheet errors. The authors of papers in the first group seem to have a mathematical or statistical background where as authors from the second group seem to have a financial background. It is quite remarkable that almost no articles can be found on spreadsheet research originating from computer scientists. Spreadsheet applications has been around since the late seventies, early eighties and are used daily by millions of people in businesses worth billions and yet very little research can be found. Besides augmenting spreadsheet systems with RTCG what else could be researched?

9 Conclusion

It can be concluded that runtime code generation (RTCG) can speed up spreadsheet calculations and the hypothesis from section 3 has been proved.

It can be concluded that the speedups seen in TinyCalc, which is written in C#, generally can be grouped into two groups. The first group giving speedup revolve around compile time type deduction, thereby avoiding type checks at runtime. It has been seen that type checks can be further divided into two subcategories, internal (for the formula expression in question) type deduction and external type deduction (for cell references). It can be concluded that both types are equally important for RTCG to be beneficial in spreadsheets. External type deduction was not implemented in this thesis, as time did not permit advanced dependency graphs to be built into TinyCalc. That said, performance measurements were conducted with an assumed working external type deduction scheme in order to get knowledge of potential speedup gained from external type deduction. It can be concluded that expressions containing cell references will benefit greatly from external type deduction as will matrix operations.

The second group of optimizations are those where careful consideration is paid to the .NET CLR and its implementation. First, it was seen that speedups could be achieved by avoiding creation of new objects in the .NET CLR. Speedup was seen when TinyCalc avoided creating new instances of **NumberValue** to hold intermediate results that just served as input the the next operation. Likewise, it was also seen that a "simple" matrix operation like `SUM(A$1:A12288)` could be speeded up a factor of nearly 4 (55 to 15 seconds) by avoiding using a `Value[,]` array to hold **NumberValue** references. And this was before any RTCG was performed.

It can also be concluded that an overhead of doing RTCG in spreadsheets *does* exist and should be taken into account. The overhead has been measured, and it can be concluded that both of the two likely candidates for doing RTCG in TinyCalc have problems. The problems are, however, not general in nature, but are caused by the way DynamicMethods and the class loader are implemented in .NET.

Comparing TinyCalc to commercial or full fledged spreadsheets like Excel, OOCalc or Gnumeric, TinyCalc has great potential. TinyCalc was consistently faster than OOCalc and Gnumeric when doing calculation by interpretation. When doing calculation using RTCG, TinyCalc approached or beat Excels calculation times. It is obvious that TinyCalc by no means is a full fledged spreadsheet, but it is interesting that a simple spreadsheet engine, written in a modern, JIT compiled language performs so well using simple structures and recalculation methods when coupled with runtime code generation.

10 References

[1] aamshukov@cogeco.ca. Ms excel grammar, bnf? `http://compilers.iecc.com/comparch/article/05-07-101`, July 2005.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Princiles, Techniques, and Tools.* Addison-Wesley, 1986.

[3] Jessica (aka JFo). Datagridview faq. `http://www.windowsforms.net/Samples/Go%20To%20Market/DataGridView/DataGridView%20FAQ.doc`, November 2005.

[4] Daniel Ballinger. Invesitgation into excel syntax and a formula grammar. `https://www.mcs.vuw.ac.nz/~db/Excel.shtml`.

[5] Inc. Borland International. Electronic spreadsheet system and methods for compiling a formula stored in a spreadsheet into native machine code for execution by a floating-point unit upon spreadsheet recalculation. United States Patent 5,471,612, November 1995.

[6] Inc. Borland International. Methods for compiling formulas stored in an electronic spreadsheet system. United States Patent 5,633,998, May 1997.

[7] Chance224. Update linked cells within a workbook??? `http://www.excelbanter.com/showthread.php?t=550&goto=nextnewest`.

[8] Microsoft Corporation David Gainer. Microsoft excel 2007 (nee excel 12) – some other numbers. `http://blogs.msdn.com/excel/archive/2005/09/26/474258.aspx`.

[9] Tobias Friedrich Deepak Ajwani and Ulrich Meyer. An $o(n^{2.75})$ algorithm for online topological ordering. `http://www.mpi-inf.mpg.de/~ajwani/ftp/SWAT06_ajwani.pdf`, 2006.

[10] Microsoft Corporation Eric Gunnerson. Calling code dynamically. `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dncscol/html/csharp02172004.asp`, May 2004.

[11] Gnome foundation. Gnumeric xml file format. `http://www.gnome.org/projects/gnumeric/doc/file-format-gnumeric.html`.

[12] David Gilbert. The gnumeric file format. `http://www.jfree.org/jworkbook/download/gnumeric-xml.pdf`, November 2001.

[13] Jody Goldberg. gnumeric-value.h source. `http://cvs.gnome.org/viewcvs/gnumeric/src/value.h?view=markup`, May 2005. Revision 1.67.

[14] Ryan Gregg. Xmldocument vs xmltextreader. http://ryangregg.com/2004/10/29/XmlDocumentVsXmlTextReaderXmlTextWriter.aspx.

[15] Stephen C. Grubb. Ploticus — a free, gpl, non-interactive software package for producing plots, charts, and graphics from data. http://ploticus.sourceforge.net/doc/welcome.html.

[16] IEEE Task P754. *ANSI/IEEE 754-1985, Standard for Binary Floating-Point Arithmetic*. IEEE, New York, August 12 1985.

[17] Ecma International. Ecma tc45 office open xml standard - draft 1.3. http://www.ecma-international.org/news/TC45_current_work/Ecma%20TC45%20OOXML%20Standard%20-%20Draft%201.3.pdf.

[18] Microsoft Corporation Joel Pobar. Reflection — dodge common performance pitfalls to craft speedy applications. MSDN Magazine, http://msdn.microsoft.com/msdnmag/issues/05/07/Reflection/default.aspx, July 2005.

[19] Mike Krueger and John Reilly. The zip, gzip, bzip2 and tar implementation for .net. http://www.icsharpcode.net/OpenSource/SharpZipLib/.

[20] Jesse Liberty. *Programming C#*. O'Reilly, 4. ed. edition, 2005.

[21] Jesse Liberty. *Visual C# 2005*. O'Reilly, 1. ed. edition, 2005.

[22] Serge Lidin. *Inside Microsoft .NET IL Assembler*. Microsoft Press, Redmond, WA, USA, 2002.

[23] Microsoft Corporation Loreen La Penna. Recalculation in microsoft excel 2002. MSDN, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnexcl2k2/html/odc_xlrecalc.asp, October 2001.

[24] Matt. Updating formula with link to another worksheet using vlookup. http://help.lockergnome.com/office/Updating-formula-link-worksheet-vlookup-ftopict639819.html.

[25] et. al. Michael Brauer. Open document format for office applications (odf) v1.0. http://docs.oasis-open.org/office/v1.0, May 2005.

[26] Microsoft Corporation Michael Stowe. Xml in excel and the spreadsheet component. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnexcl2k2/html/odc_xlsmlinss.asp, August 2001. 8 printed pages.

[27] Microsoft Corporation Michael Stowe. Xml spreadsheet reference. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnexcl2k2/html/odc_xmlss.asp, August 2001. 53 printed pages.

[28] Microsoft Corporation Mike Stall. Debugging dynamically generated code (reflection.emit). Mike Stall's .NET Debugging Blog, `http://blogs.msdn.com/jmstall/archive/2005/02/03/366429.aspx`, February 2005.

[29] Microsoft Corporation Mike Stall. Mdbg extension to debug il. Mike Stall's .NET Debugging Blog, `http://blogs.msdn.com/jmstall/archive/2005/11/04/mdbg_il_debugging.aspx`, October 2005.

[30] Decision Models. Excel pages - calculation secrets. `http://www.decisionmodels.com/calcsecrets.htm`.

[31] Glenford J. Myers. *Art of Software Testing.* John Wiley & Sons, Inc., New York, NY, USA, 1979.

[32] David J. Pearce and Paul H. J. Kelly. Online algorithms for maintaining the topological order of a directed acyclic graph. `http://www.mcs.vuw.ac.nz/~djp/files/tr0703.pdf`, July 2003.

[33] Department of Mathematics Peter Sestoft, Royal Veterinary Physics, and Copenhagen Denmark & IT University of Copenhagen Agricultural University. Runtime code generation with jvm and clr. `http://www.dina.kvl.dk/~sestoft/rtcg/rtcg.pdf`, October 2002.

[34] Peter Sestoft and Henrik I. Hansen. *C# precisely.* MIT Pr., 2004.

[35] Microsoft Corporation Unknown. Info: Microsoft excel 2002 and xml. `http://support.microsoft.com/kb/288215/EN-US/`, December 2003. Revision 1.0.

[36] Haibo Luo's weblog. Debuggervisualizer for dynamicmethod (show me the il). Haibo Luo's weblog, `http://blogs.msdn.com/haibo_luo/archive/2005/10/25/484861.aspx`, October 2005.

A Appendix

A.1 About this thesis

For those interested a short description of this thesis is presented.

The thesis is written in LaTeX, which is a typesetting system developed by Donald E. Knuth for his wife. As the code for TinyCalc is written in Visual Studio 2005, the MikTex package for Windows is used. To edit the actual `.tex` files, WinEdt a versatile LaTeX editor, was used.

The layout used is the article layout of LaTeX on A4 paper with 12 point fonts, wide margins (`a4wide` package) and a line spacing 1.5. Code listings are produced using a simple `verbatim` environment. Internet references are produced using the `URL` package, the `hyperref` package takes care of producing clickable links in the PDF file, graphs and charts are constructed automatically using **ploticus** and finally the `fixme` package is used for keeping track of things needed to be fixed before publishing a document.

A.2 Recalculation loop expressed as pseudocode

The three pieces of pseudo code presented below performs the recalculation in TinyCalc for RTCG.

```
Workbook.Recompute() {
   // Before doing a recomputation of the entire workbook
   // Clear the compiler dictionary (used for formulasharing).
   if (GeneratorOptions.Level >= GeneratorLevel.Level2)
      RTCGDict.Clear();

   cyclic = false;

   // O(1) way of marking all formulas
   // non-visited and non-uptodate.
   set = !set;

   // Now for all cached expressions, ce.visited != set and ce.uptodate != set
   try
   {
      foreach (Sheet sheet in sheets)
         sheet.Recompute();
   }
   catch (Cyclic)
   {
      foreach (Sheet sheet in sheets)
         sheet.Reset();
      cyclic = true;
      throw;
   }
}

sheet.Recompute() {
   for (int col = 0; col < Cols; col++)
      for (int row = 0; row < Rows; row++)
      {
         Cell cell = cells[col, row];
         if (cell != null)
            cell.Eval(this, col, row);
      }
}


cell.Eval (for Functions) {
   if (uptodate != workbook.Set)
   {
      if (visited == workbook.Set)
         throw new Cyclic("Cyclic cell reference: ");
      else
      {
         visited = workbook.Set;
         switch (GeneratorOptions.Level)
         {
            case GeneratorLevel.Level0:
            default:
               v = Level0_Eval(sheet, col, row);
               break;
            case GeneratorLevel.Level1:
               v = Level1_GenEval(sheet, col, row);
               break;
            case GeneratorLevel.Level2:
            case GeneratorLevel.Level3:
            case GeneratorLevel.Level4:
            case GeneratorLevel.Level5:
            case GeneratorLevel.Level6:
            case GeneratorLevel.Level7:
               v = Level2P_GenEval(sheet, col, row);
               break;
         }
         uptodate = workbook.Set;
      }
   }
   return v;
}

Level23_GenEval (for functions) {
   if (GeneratorOptions.UseFormulaSharing)
   {
      rtcgexpr = RTCGDict.Lookup(e);
      if (rtcgexpr == null)
      {
         rtcgexpr = e.InlineSubExpr_Gen(sheet, col, row);
         RTCGDict.Insert(e, rtcgexpr);
      }
   }
   else
   {
      rtcgexpr = e.InlineSubExpr_Gen(sheet, col, row);
   }

   if (rtcgexpr != null)
      return rtcgexpr.Eval(sheet, col, row);
   else
      return null;
}
```

A.3 BNF grammar for formula expressions in TinyCalc

```
/* Coco/R grammar for spreadsheet formulas */

/* mono ~/cs/coco/Coco.exe Spreadsheet.ATG -namespace Spreadsheet */
/* gmcs Spreadsheet.cs Scanner.cs Parser.cs */


/* Originally written by Peter Sestoft, 2005 */
/* Modified by Thomas S. Iversen, January, April 2006 to support: */

/* - RaRefs in the R1C1 format
 * - The string concatnation operator &
 * - Numbers in scientific notation
 * - Sheetreferences in the style: [Alpha{Alpha}!]Raref
 * - ^ (power) operator (April 2006).
 *
 * Compile with: coco -namespace TinyCalc Spreadsheet.ATG
 */

using System.Collections;

COMPILER Spreadsheet

  private int col, row;
  private Workbook workbook;
  private Cell cell;
  System.Globalization.NumberFormatInfo ni = null;

  public void SetNumberDecimalSeparator(String nds) {
         System.Globalization.CultureInfo ci =
            System.Globalization.CultureInfo.InstalledUICulture;
         this.ni = (System.Globalization.NumberFormatInfo)
            ci.NumberFormat.Clone();
         this.ni.NumberDecimalSeparator = nds;
  }


  public Cell ParseCell(Workbook workbook, int col, int row) {
    this.workbook = workbook;
    this.col = col; this.row = row;
    // US decimal point, regardless of culture
    SetNumberDecimalSeparator(".");
    Parse();
    return cell;
  }

/*------------------------------------------------------------------------*/
CHARACTERS
  letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".
  atoi   = "ABCDEFGHIabcdefghi".
  digit = "0123456789".
  Alpha = letter + digit.
  cr  = ’\r’.
  lf  = ’\n’.
  tab = ’\t’.
  exclamation = ’!’.
  dollar  = ’$’.
  newLine = cr + lf.
  strchar    = ANY - ’"’ - ’\\’ - newLine.
  char       = ANY - ’\\’ - newLine.


TOKENS
  name       = letter { letter } CONTEXT("(").
  number     =
               digit { digit }
               [                                 /* optional fraction */
                [ "." digit { digit }]     /* optional fractional digits */
                [ ( "E" | "e" )                  /* optional exponent */
                  [ "+" | "-" ]                  /* optinoal exponentsign */
                  digit { digit }
                ]
               ].
  sheetref   = Alpha { Alpha } exclamation.
  raref      = [ dollar ] atoi { letter } [ dollar ] digit { digit }.
  xmlssraref1= "RC".
  xmlssraref2= "R" digit {digit} "C".
  xmlssraref3= "R[" ["+"|"-"] digit { digit } "]C".
  xmlssraref4= "R" digit {digit} "C" digit {digit}.
  xmlssraref5= "R[" ["+"|"-"] digit { digit } "]C" digit {digit}.
  xmlssraref6= "RC" digit {digit}.
  xmlssraref7= "RC[" ["+"|"-"] digit { digit } "]".
  xmlssraref8= "R[" ["+"|"-"] digit { digit } "]C[" ["+"|"-"] digit { digit } "]".
  string     = "\"" { strchar } "\"".
  textcell   = "\’" { char }.

COMMENTS FROM "/*" TO "*/" NESTED
COMMENTS FROM "//" TO cr lf

IGNORE cr + lf + tab


PRODUCTIONS
/*------------------------------------------------------------------------*/
AddOp<out String op>
=                        (. op = "+"; .)
  ( ’+’
  | ’-’                  (. op = "-"; .)
  | ’&’                  (. op = "&"; .)
  ).

LogicalOp<out String op>
=                        (. op = "=="; .)
  ( "=="
  | "<>"                 (. op = "<>"; .)
  | "<"                  (. op = "<";  .)
  | "<="                 (. op = "<="; .)
  | ">"                  (. op = ">";  .)
  | ">="                 (. op = ">="; .)
  ).

/*------------------------------------------------------------------------*/
Expr<out Expr e>         (. Expr e2; String op; e = null; .)
= LogicalTerm<out e>
  { LogicalOp<out op>
    LogicalTerm<out e2>  (. e = new FunCall(op, new Expr[] { e, e2 }); .)
  }
  .


LogicalTerm<out Expr e>  (. Expr e2; String op; e = null; .)
= Term<out e>
```

```
  { AddOp<out op>
    Term<out e2>         (. e = new FunCall(op, new Expr[] { e, e2 }); .)
  }
  .


/*------------------------------------------------------------------------*/
Factor<out Expr e>       (. RARef r1, r2; Sheet s1 = null; double d; e = null; .)
= (
   | sheetref            (.
                                String sheetname = t.val.TrimEnd("!".ToCharArray());
                                s1 = workbook[sheetname];
                        .)
  )
   Raref<out r1> (       (. e = new CellRef(s1, r1);      .)
    | ':' Raref<out r2>  (. e = new CellArea(null, r1, r2); .)
   )
  | Number<out d>        (. e = new NumberConst(d);         .)
  | string               (. int len = t.val.Length-2;
                            e = new TextConst(t.val.Substring(1, len));
                        .)
  | '(' Expr<out e> ')'
  | Application<out e>
  .

/*------------------------------------------------------------------------*/
PowFactor<out Expr e>    (. Expr e2; .)
= Factor<out e>
  { '^'
    Factor<out e2>       (. e = new FunCall("^", new Expr[] { e, e2 } ); .)
  }
  .
/*------------------------------------------------------------------------*/

Raref<out RARef raref>   (. raref = null;.)
= raref                  (. raref = new RARef(t.val, col, row); .)
 | xmlssraref1           (. raref = new RARef(t.val); .)
 | xmlssraref2           (. raref = new RARef(t.val); .)
 | xmlssraref3           (. raref = new RARef(t.val); .)
 | xmlssraref4           (. raref = new RARef(t.val); .)
 | xmlssraref5           (. raref = new RARef(t.val); .)
 | xmlssraref6            (. raref = new RARef(t.val); .)
 | xmlssraref7            (. raref = new RARef(t.val); .)
 | xmlssraref8            (. raref = new RARef(t.val); .)
.

/*----------------------------------------------------------------------------*/
Number<out double d>     (. d = 0.0;                      .)
= ( number               (. d = double.Parse(t.val, ni); .)
  | ’-’ number           (. d = -double.Parse(t.val, ni); .)
  ) .
/*----------------------------------------------------------------------------*/
Application<out Expr e>  (. String s; Expr[] es; e = null; .)
= Name<out s> ’(’
  ( ’)’                  (. e = new FunCall(s, new Expr[0]); .)
    | Exprs1<out es> ’)’ (. e = new FunCall(s, es); .)
  )
.
/*----------------------------------------------------------------------------*/
Exprs1<out Expr[] es>    (. Expr e1, e2;
                            ArrayList elist = new ArrayList();
                        .)
= ( Expr<out e1>         (. elist.Add(e1); .)
    { (’;’ | ’,’) Expr<out e2>   (. elist.Add(e2); .)
    }
  )                      (. es = new Expr[elist.Count];
                            elist.CopyTo(es, 0);
                        .)
.
/*----------------------------------------------------------------------------*/
Name<out String s>
= name                   (. s = t.val; .)
.
/*----------------------------------------------------------------------------*/
MulOp<out String op>
=                        (. op = "*"; .)
  ( ’*’
  | ’/’                  (. op = "/"; .)
  ).
/*----------------------------------------------------------------------------*/
Term<out Expr e>         (. Expr e2; String op; .)
= PowFactor<out e>
```

```
  { MulOp<out op>
    PowFactor<out e2>    (. e = new FunCall(op, new Expr[] { e, e2 } ); .)
  }.
/*------------------------------------------------------------------------*/
Spreadsheet             (. Expr e; double d;     .)
= ( '=' Expr<out e>     (. this.cell = new Formula(workbook, e); .)
  | textcell            (. this.cell = new TextCell(t.val.Substring(1)); .)
  | Number<out d>       (. this.cell = new NumberCell(d);  .)
  | string              (. this.cell = new TextCell(t.val.Substring(1)); .)

  ).

END Spreadsheet.
```

A.4 Overhead of doing RTCG

A.4.1 The general power function in C#

Example 14 A C# version of the general x^y function

```
public int pow(int x, int y) {
    int r = 1;
    while(y > 0) {
        if(y % 2 == 0) {
            x = x * x;
            y = y / 2;
        } else {
            r = r * x;
            y = y - 1;
        }
    }
    return r;
}
```

A.4.2 The general power function in IL

Example 15 A IL version of the general power function x^y

```
// Assume that first local variable is double x
// Assume that second local variable is int y

Label elsepart = ilg.DefineLabel();
Label looptest = ilg.DefineLabel();
Label loopstart = ilg.DefineLabel();

// Declare and initilize r to 1 as a local variable
LocalBuilder r = ilg.DeclareLocal(typeof(double));
ilg.Emit(OpCodes.Ldc_R8, 1.0);
ilg.Emit(OpCodes.Stloc, r);

// Jump to the test of the while loop exit condition
ilg.Emit(OpCodes.Br_S, looptest);

// y % 2 == 0?
ilg.MarkLabel(loopstart);
ilg.Emit(OpCodes.Ldarg, yarg);
ilg.Emit(OpCodes.Ldc_I4_2);
ilg.Emit(OpCodes.Rem);
ilg.Emit(OpCodes.Ldc_I4_0);
ilg.Emit(OpCodes.Ceq);
ilg.Emit(OpCodes.Brfalse_S, elsepart);

// Yes, x = x * x
ilg.Emit(OpCodes.Ldarg, xarg);
ilg.Emit(OpCodes.Ldarg, xarg);
ilg.Emit(OpCodes.Mul);
ilg.Emit(OpCodes.Starg_S, xarg);

// y = y / 2
ilg.Emit(OpCodes.Ldarg, yarg);
ilg.Emit(OpCodes.Ldc_I4_2);
ilg.Emit(OpCodes.Div);
ilg.Emit(OpCodes.Starg_S, yarg);
ilg.Emit(OpCodes.Br_S, looptest);

// No, r = r * x
ilg.MarkLabel(elsepart);
ilg.Emit(OpCodes.Ldloc, r);
ilg.Emit(OpCodes.Ldarg, xarg);
ilg.Emit(OpCodes.Mul);
ilg.Emit(OpCodes.Stloc, r);

// y = y - 1
ilg.Emit(OpCodes.Ldarg, yarg);
ilg.Emit(OpCodes.Ldc_I4_1);
ilg.Emit(OpCodes.Sub);
ilg.Emit(OpCodes.Starg_S, yarg);

// y > 0?
ilg.MarkLabel(looptest);
ilg.Emit(OpCodes.Ldarg, yarg);
ilg.Emit(OpCodes.Ldc_I4_0);
ilg.Emit(OpCodes.Cgt);
ilg.Emit(OpCodes.Brtrue_S, loopstart);

// return r;
ilg.Emit(OpCodes.Ldloc, r);
ilg.Emit(OpCodes.Ret);
```

A.5 IL Examples

This section will present the IL code generated when going through the optimization levels for a couple of expressions:

A.5.1 Level 2 for `A1=5+6+7`

```
IL_0000: /* 02  |          */ ldarg.0
IL_0001: /* 7b  | 04000002 */ ldfld      TinyCalc.NumberValue value/TinyCalc.NumberConst
IL_0006: /* 75  | 02000003 */ isinst     TinyCalc.NumberValue
IL_000b: /* 0c  |          */ stloc.2
IL_000c: /* 02  |          */ ldarg.0
IL_000d: /* 7b  | 04000004 */ ldfld      TinyCalc.NumberValue value/TinyCalc.NumberConst
IL_0012: /* 75  | 02000005 */ isinst     TinyCalc.NumberValue
IL_0017: /* 0d  |          */ stloc.3
IL_0018: /* 08  |          */ ldloc.2
IL_0019: /* 39  | 0000001A */ brfalse    IL_0038
IL_001e: /* 09  |          */ ldloc.3
IL_001f: /* 39  | 00000014 */ brfalse    IL_0038
IL_0024: /* 08  |          */ ldloc.2
IL_0025: /* 7b  | 04000006 */ ldfld      Double value/TinyCalc.NumberValue
IL_002a: /* 09  |          */ ldloc.3
IL_002b: /* 7b  | 04000007 */ ldfld      Double value/TinyCalc.NumberValue
IL_0030: /* 58  |          */ add
IL_0031: /* 73  | 06000008 */ newobj     Void .ctor(Double)/TinyCalc.NumberValue
IL_0036: /* 2b  | 0A       */ br.s       IL_0042
IL_0038: /* 72  | 70000009 */ ldstr      "ARGTYPE"
IL_003d: /* 73  | 0600000A */ newobj     Void .ctor(System.String)/TinyCalc.ErrorValue
IL_0042: /* 75  | 0200000B */ isinst     TinyCalc.NumberValue
IL_0047: /* 0a  |          */ stloc.0
IL_0048: /* 02  |          */ ldarg.0
IL_0049: /* 7b  | 0400000C */ ldfld      TinyCalc.NumberValue value/TinyCalc.NumberConst
IL_004e: /* 75  | 0200000D */ isinst     TinyCalc.NumberValue
IL_0053: /* 0b  |          */ stloc.1
IL_0054: /* 06  |          */ ldloc.0
IL_0055: /* 39  | 0000001A */ brfalse    IL_0074
IL_005a: /* 07  |          */ ldloc.1
IL_005b: /* 39  | 00000014 */ brfalse    IL_0074
IL_0060: /* 06  |          */ ldloc.0
IL_0061: /* 7b  | 0400000E */ ldfld      Double value/TinyCalc.NumberValue
IL_0066: /* 07  |          */ ldloc.1
IL_0067: /* 7b  | 0400000F */ ldfld      Double value/TinyCalc.NumberValue
IL_006c: /* 58  |          */ add
IL_006d: /* 73  | 06000010 */ newobj     Void .ctor(Double)/TinyCalc.NumberValue
IL_0072: /* 2b  | 0A       */ br.s       IL_007e
IL_0074: /* 72  | 70000011 */ ldstr      "ARGTYPE"
IL_0079: /* 73  | 06000012 */ newobj     Void .ctor(System.String)/TinyCalc.ErrorValue
IL_007e: /* 2a  |          */ ret
```

A.5.2 Level 3 for `A1=5+6+7`

```
IL_0000: /* 02  |          */ ldarg.0
IL_0001: /* 7b  | 04000002 */ ldfld      TinyCalc.NumberValue value/TinyCalc.NumberConst
IL_0006: /* 0c  |          */ stloc.2
IL_0007: /* 02  |          */ ldarg.0
IL_0008: /* 7b  | 04000003 */ ldfld      TinyCalc.NumberValue value/TinyCalc.NumberConst
IL_000d: /* 0d  |          */ stloc.3
IL_000e: /* 08  |          */ ldloc.2
IL_000f: /* 7b  | 04000004 */ ldfld      Double value/TinyCalc.NumberValue
IL_0014: /* 09  |          */ ldloc.3
IL_0015: /* 7b  | 04000005 */ ldfld      Double value/TinyCalc.NumberValue
IL_001a: /* 58  |          */ add
IL_001b: /* 73  | 06000006 */ newobj     Void .ctor(Double)/TinyCalc.NumberValue
IL_0020: /* 0a  |          */ stloc.0
IL_0021: /* 02  |          */ ldarg.0
IL_0022: /* 7b  | 04000007 */ ldfld      TinyCalc.NumberValue value/TinyCalc.NumberConst
IL_0027: /* 0b  |          */ stloc.1
IL_0028: /* 06  |          */ ldloc.0
IL_0029: /* 7b  | 04000008 */ ldfld      Double value/TinyCalc.NumberValue
IL_002e: /* 07  |          */ ldloc.1
IL_002f: /* 7b  | 04000009 */ ldfld      Double value/TinyCalc.NumberValue
IL_0034: /* 58  |          */ add
IL_0035: /* 73  | 0600000A */ newobj     Void .ctor(Double)/TinyCalc.NumberValue
IL_003a: /* 2a  |          */ ret
```

A.5.3 Level 4 for `A1=5+6+7`

```
IL_0000: /* 02  |          */ ldarg.0
IL_0001: /* 7b  | 04000002 */ ldfld      TinyCalc.NumberValue value/TinyCalc.NumberConst
IL_0006: /* 13  | 04       */ stloc.s    V_4
IL_0008: /* 02  |          */ ldarg.0
IL_0009: /* 7b  | 04000003 */ ldfld      TinyCalc.NumberValue value/TinyCalc.NumberConst
IL_000e: /* 13  | 05       */ stloc.s    V_5
IL_0010: /* 11  | 04       */ ldloc.s    V_4
IL_0012: /* 7b  | 04000004 */ ldfld      Double value/TinyCalc.NumberValue
IL_0017: /* 11  | 05       */ ldloc.s    V_5
IL_0019: /* 7b  | 04000005 */ ldfld      Double value/TinyCalc.NumberValue
IL_001e: /* 58  |          */ add
IL_001f: /* 0c  |          */ stloc.2
IL_0020: /* 02  |          */ ldarg.0
IL_0021: /* 7b  | 04000006 */ ldfld      TinyCalc.NumberValue value/TinyCalc.NumberConst
IL_0026: /* 0b  |          */ stloc.1
IL_0027: /* 08  |          */ ldloc.2
IL_0028: /* 07  |          */ ldloc.1
IL_0029: /* 7b  | 04000007 */ ldfld      Double value/TinyCalc.NumberValue
IL_002e: /* 58  |          */ add
IL_002f: /* 73  | 06000008 */ newobj     Void .ctor(Double)/TinyCalc.NumberValue
IL_0034: /* 2a  |          */ ret
```

A.5.4 Level 5 for A1=5+6+7

```
IL_0000: /* 23  | 5        */ ldc.r8     5
IL_0009: /* 13  | 06       */ stloc.s    V_6
IL_000b: /* 23  | 6        */ ldc.r8     6
IL_0014: /* 13  | 07       */ stloc.s    V_7
IL_0016: /* 11  | 06       */ ldloc.s    V_6
IL_0018: /* 11  | 07       */ ldloc.s    V_7
IL_001a: /* 58  |          */ add
IL_001b: /* 0c  |          */ stloc.2
IL_001c: /* 23  | 7        */ ldc.r8     7
IL_0025: /* 0d  |          */ stloc.3
IL_0026: /* 08  |          */ ldloc.2
IL_0027: /* 09  |          */ ldloc.3
IL_0028: /* 58  |          */ add
IL_0029: /* 73  | 06000002 */ newobj     Void .ctor(Double)/TinyCalc.NumberValue
IL_002e: /* 2a  |          */ ret
```

A.5.5 Level 6 for A1=5+6+7

```
IL_0000: /* 23  | 5        */ ldc.r8     5
IL_0009: /* 23  | 6        */ ldc.r8     6
IL_0012: /* 58  |          */ add
IL_0013: /* 23  | 7        */ ldc.r8     7
IL_001c: /* 58  |          */ add
IL_001d: /* 73  | 06000002 */ newobj     Void .ctor(Double)/TinyCalc.NumberValue
IL_0022: /* 2a  |          */ ret
```

A.5.6 Level 7 for A1=5+6+7

```
IL_0000: /* 23  | 5        */ ldc.r8     5
IL_0009: /* 13  | 06       */ stloc.s    V_6
IL_000b: /* 23  | 6        */ ldc.r8     6
IL_0014: /* 13  | 07       */ stloc.s    V_7
IL_0016: /* 11  | 06       */ ldloc.s    V_6
IL_0018: /* 11  | 07       */ ldloc.s    V_7
IL_001a: /* 58  |          */ add
IL_001b: /* 0c  |          */ stloc.2
IL_001c: /* 23  | 7        */ ldc.r8     7
IL_0025: /* 0d  |          */ stloc.3
IL_0026: /* 08  |          */ ldloc.2
IL_0027: /* 09  |          */ ldloc.3
IL_0028: /* 58  |          */ add
IL_0029: /* 73  | 06000002 */ newobj     Void .ctor(Double)/TinyCalc.NumberValue
IL_002e: /* 2a  |          */ ret
```

A.5.7 Level 2-3 for `A1=A2+A3+A4`

```
IL_0000: /* 20  | 00000000 */ ldc.i4     0
IL_0005: /* 0e  | 02       */ ldarg.s    V_2
IL_0007: /* 58  |          */ add
IL_0008: /* 13  | 06       */ stloc.s    V_6
IL_000a: /* 20  | 00000001 */ ldc.i4     1
IL_000f: /* 0e  | 03       */ ldarg.s    V_3
IL_0011: /* 58  |          */ add
IL_0012: /* 13  | 07       */ stloc.s    V_7
IL_0014: /* 0e  | 01       */ ldarg.s    V_1
IL_0016: /* 11  | 06       */ ldloc.s    V_6
IL_0018: /* 11  | 07       */ ldloc.s    V_7
IL_001a: /* 6f  | 0A000002 */ callvirt   TinyCalc.Cell get_Item(Int32, Int32)/TinyCalc.Sheet
IL_001f: /* 13  | 04       */ stloc.s    V_4
IL_0021: /* 11  | 04       */ ldloc.s    V_4
IL_0023: /* 2c  | 0F       */ brfalse.s  IL_0034
IL_0025: /* 11  | 04       */ ldloc.s    V_4
IL_0027: /* 0e  | 01       */ ldarg.s    V_1
IL_0029: /* 11  | 06       */ ldloc.s    V_6
IL_002b: /* 11  | 07       */ ldloc.s    V_7
IL_002d: /* 6f  | 0A000003 */ callvirt   TinyCalc.Value Eval(TinyCalc.Sheet, Int32, Int32)/TinyCalc.Cell
IL_0032: /* 2b  | 01       */ br.s       IL_0035
IL_0034: /* 14  |          */ ldnull
IL_0035: /* 13  | 05       */ stloc.s    V_5
IL_0037: /* 11  | 05       */ ldloc.s    V_5
IL_0039: /* 75  | 02000004 */ isinst     TinyCalc.NumberValue
IL_003e: /* 0c  |          */ stloc.2
IL_003f: /* 20  | 00000000 */ ldc.i4     0
IL_0044: /* 0e  | 02       */ ldarg.s    V_2
IL_0046: /* 58  |          */ add
IL_0047: /* 13  | 0A       */ stloc.s    V_10
IL_0049: /* 20  | 00000002 */ ldc.i4     2
IL_004e: /* 0e  | 03       */ ldarg.s    V_3
IL_0050: /* 58  |          */ add
IL_0051: /* 13  | 0B       */ stloc.s    V_11
IL_0053: /* 0e  | 01       */ ldarg.s    V_1
IL_0055: /* 11  | 0A       */ ldloc.s    V_10
IL_0057: /* 11  | 0B       */ ldloc.s    V_11
IL_0059: /* 6f  | 0A000005 */ callvirt   TinyCalc.Cell get_Item(Int32, Int32)/TinyCalc.Sheet
IL_005e: /* 13  | 08       */ stloc.s    V_8
IL_0060: /* 11  | 08       */ ldloc.s    V_8
IL_0062: /* 2c  | 0F       */ brfalse.s  IL_0073
IL_0064: /* 11  | 08       */ ldloc.s    V_8
IL_0066: /* 0e  | 01       */ ldarg.s    V_1
IL_0068: /* 11  | 0A       */ ldloc.s    V_10
IL_006a: /* 11  | 0B       */ ldloc.s    V_11
IL_006c: /* 6f  | 0A000006 */ callvirt   TinyCalc.Value Eval(TinyCalc.Sheet, Int32, Int32)/TinyCalc.Cell
IL_0071: /* 2b  | 01       */ br.s       IL_0074
IL_0073: /* 14  |          */ ldnull
IL_0074: /* 13  | 09       */ stloc.s    V_9
IL_0076: /* 11  | 09       */ ldloc.s    V_9
IL_0078: /* 75  | 02000007 */ isinst     TinyCalc.NumberValue
```

```
IL_007d: /* 0d  |          */ stloc.3
IL_007e: /* 08  |          */ ldloc.2
IL_007f: /* 39  | 0000001A */ brfalse    IL_009e
IL_0084: /* 09  |          */ ldloc.3
IL_0085: /* 39  | 00000014 */ brfalse    IL_009e
IL_008a: /* 08  |          */ ldloc.2
IL_008b: /* 7b  | 04000008 */ ldfld      Double value/TinyCalc.NumberValue
IL_0090: /* 09  |          */ ldloc.3
IL_0091: /* 7b  | 04000009 */ ldfld      Double value/TinyCalc.NumberValue
IL_0096: /* 58  |          */ add
IL_0097: /* 73  | 0600000A */ newobj     Void .ctor(Double)/TinyCalc.NumberValue
IL_009c: /* 2b  | 0A       */ br.s       IL_00a8
IL_009e: /* 72  | 7000000B */ ldstr      "ARGTYPE"
IL_00a3: /* 73  | 0600000C */ newobj     Void .ctor(System.String)/TinyCalc.ErrorValue
IL_00a8: /* 75  | 0200000D */ isinst     TinyCalc.NumberValue
IL_00ad: /* 0a  |          */ stloc.0
IL_00ae: /* 20  | 00000000 */ ldc.i4     0
IL_00b3: /* 0e  | 02       */ ldarg.s    V_2
IL_00b5: /* 58  |          */ add
IL_00b6: /* 13  | 0E       */ stloc.s    V_14
IL_00b8: /* 20  | 00000003 */ ldc.i4     3
IL_00bd: /* 0e  | 03       */ ldarg.s    V_3
IL_00bf: /* 58  |          */ add
IL_00c0: /* 13  | 0F       */ stloc.s    V_15
IL_00c2: /* 0e  | 01       */ ldarg.s    V_1
IL_00c4: /* 11  | 0E       */ ldloc.s    V_14
IL_00c6: /* 11  | 0F       */ ldloc.s    V_15
IL_00c8: /* 6f  | 0A00000E */ callvirt   TinyCalc.Cell get_Item(Int32, Int32)/TinyCalc.Sheet
IL_00cd: /* 13  | 0C       */ stloc.s    V_12
IL_00cf: /* 11  | 0C       */ ldloc.s    V_12
IL_00d1: /* 2c  | 0F       */ brfalse.s  IL_00e2
IL_00d3: /* 11  | 0C       */ ldloc.s    V_12
IL_00d5: /* 0e  | 01       */ ldarg.s    V_1
IL_00d7: /* 11  | 0E       */ ldloc.s    V_14
IL_00d9: /* 11  | 0F       */ ldloc.s    V_15
IL_00db: /* 6f  | 0A00000F */ callvirt   TinyCalc.Value Eval(TinyCalc.Sheet, Int32, Int32)/TinyCalc.Cell
IL_00e0: /* 2b  | 01       */ br.s       IL_00e3
IL_00e2: /* 14  |          */ ldnull
IL_00e3: /* 13  | 0D       */ stloc.s    V_13
IL_00e5: /* 11  | 0D       */ ldloc.s    V_13
IL_00e7: /* 75  | 02000010 */ isinst     TinyCalc.NumberValue
IL_00ec: /* 0b  |          */ stloc.1
IL_00ed: /* 06  |          */ ldloc.0
IL_00ee: /* 39  | 0000001A */ brfalse    IL_010d
IL_00f3: /* 07  |          */ ldloc.1
IL_00f4: /* 39  | 00000014 */ brfalse    IL_010d
IL_00f9: /* 06  |          */ ldloc.0
IL_00fa: /* 7b  | 04000011 */ ldfld      Double value/TinyCalc.NumberValue
IL_00ff: /* 07  |          */ ldloc.1
IL_0100: /* 7b  | 04000012 */ ldfld      Double value/TinyCalc.NumberValue
IL_0105: /* 58  |          */ add
IL_0106: /* 73  | 06000013 */ newobj     Void .ctor(Double)/TinyCalc.NumberValue
```

```
IL_010b: /* 2b  | 0A       */ br.s       IL_0117
IL_010d: /* 72  | 70000014 */ ldstr      "ARGTYPE"
IL_0112: /* 73  | 06000015 */ newobj     Void .ctor(System.String)/TinyCalc.ErrorValue
IL_0117: /* 2a  |          */ ret
```

A.5.8 Level 4-5 for A1=A2+A3+A4

```
IL_0000: /* 20  | 00000000 */ ldc.i4     0
IL_0005: /* 0e  | 02       */ ldarg.s    V_2
IL_0007: /* 58  |          */ add
IL_0008: /* 13  | 0A       */ stloc.s    V_10
IL_000a: /* 20  | 00000001 */ ldc.i4     1
IL_000f: /* 0e  | 03       */ ldarg.s    V_3
IL_0011: /* 58  |          */ add
IL_0012: /* 13  | 0B       */ stloc.s    V_11
IL_0014: /* 0e  | 01       */ ldarg.s    V_1
IL_0016: /* 11  | 0A       */ ldloc.s    V_10
IL_0018: /* 11  | 0B       */ ldloc.s    V_11
IL_001a: /* 6f  | 0A000002 */ callvirt   TinyCalc.Cell get_Item(Int32, Int32)/TinyCalc.Sheet
IL_001f: /* 13  | 08       */ stloc.s    V_8
IL_0021: /* 11  | 08       */ ldloc.s    V_8
IL_0023: /* 2c  | 0F       */ brfalse.s  IL_0034
IL_0025: /* 11  | 08       */ ldloc.s    V_8
IL_0027: /* 0e  | 01       */ ldarg.s    V_1
IL_0029: /* 11  | 0A       */ ldloc.s    V_10
IL_002b: /* 11  | 0B       */ ldloc.s    V_11
IL_002d: /* 6f  | 0A000003 */ callvirt   TinyCalc.Value Eval(TinyCalc.Sheet, Int32, Int32)/TinyCalc.Cell
IL_0032: /* 2b  | 01       */ br.s       IL_0035
IL_0034: /* 14  |          */ ldnull
IL_0035: /* 13  | 09       */ stloc.s    V_9
IL_0037: /* 11  | 09       */ ldloc.s    V_9
IL_0039: /* 75  | 02000004 */ isinst     TinyCalc.NumberValue
IL_003e: /* 13  | 04       */ stloc.s    V_4
IL_0040: /* 20  | 00000000 */ ldc.i4     0
IL_0045: /* 0e  | 02       */ ldarg.s    V_2
IL_0047: /* 58  |          */ add
IL_0048: /* 13  | 0E       */ stloc.s    V_14
IL_004a: /* 20  | 00000002 */ ldc.i4     2
IL_004f: /* 0e  | 03       */ ldarg.s    V_3
IL_0051: /* 58  |          */ add
IL_0052: /* 13  | 0F       */ stloc.s    V_15
IL_0054: /* 0e  | 01       */ ldarg.s    V_1
IL_0056: /* 11  | 0E       */ ldloc.s    V_14
IL_0058: /* 11  | 0F       */ ldloc.s    V_15
IL_005a: /* 6f  | 0A000005 */ callvirt   TinyCalc.Cell get_Item(Int32, Int32)/TinyCalc.Sheet
IL_005f: /* 13  | 0C       */ stloc.s    V_12
IL_0061: /* 11  | 0C       */ ldloc.s    V_12
IL_0063: /* 2c  | 0F       */ brfalse.s  IL_0074
IL_0065: /* 11  | 0C       */ ldloc.s    V_12
IL_0067: /* 0e  | 01       */ ldarg.s    V_1
IL_0069: /* 11  | 0E       */ ldloc.s    V_14
IL_006b: /* 11  | 0F       */ ldloc.s    V_15
```

```
IL_006d: /* 6f  | 0A000006 */ callvirt   TinyCalc.Value Eval(TinyCalc.Sheet, Int32, Int32)/TinyCalc.Cell
IL_0072: /* 2b  | 01       */ br.s       IL_0075
IL_0074: /* 14  |          */ ldnull
IL_0075: /* 13  | 0D       */ stloc.s    V_13
IL_0077: /* 11  | 0D       */ ldloc.s    V_13
IL_0079: /* 75  | 02000007 */ isinst     TinyCalc.NumberValue
IL_007e: /* 13  | 05       */ stloc.s    V_5
IL_0080: /* 11  | 04       */ ldloc.s    V_4
IL_0082: /* 39  | 0000001D */ brfalse    IL_00a4
IL_0087: /* 11  | 05       */ ldloc.s    V_5
IL_0089: /* 39  | 00000016 */ brfalse    IL_00a4
IL_008e: /* 11  | 04       */ ldloc.s    V_4
IL_0090: /* 7b  | 04000008 */ ldfld      Double value/TinyCalc.NumberValue
IL_0095: /* 11  | 05       */ ldloc.s    V_5
IL_0097: /* 7b  | 04000009 */ ldfld      Double value/TinyCalc.NumberValue
IL_009c: /* 58  |          */ add
IL_009d: /* 73  | 0600000A */ newobj     Void .ctor(Double)/TinyCalc.NumberValue
IL_00a2: /* 2b  | 0A       */ br.s       IL_00ae
IL_00a4: /* 72  | 7000000B */ ldstr      "ARGTYPE"
IL_00a9: /* 73  | 0600000C */ newobj     Void .ctor(System.String)/TinyCalc.ErrorValue
IL_00ae: /* 75  | 0200000D */ isinst     TinyCalc.NumberValue
IL_00b3: /* 0a  |          */ stloc.0
IL_00b4: /* 20  | 00000000 */ ldc.i4     0
IL_00b9: /* 0e  | 02       */ ldarg.s    V_2
IL_00bb: /* 58  |          */ add
IL_00bc: /* 13  | 12       */ stloc.s    V_18
IL_00be: /* 20  | 00000003 */ ldc.i4     3
IL_00c3: /* 0e  | 03       */ ldarg.s    V_3
IL_00c5: /* 58  |          */ add
IL_00c6: /* 13  | 13       */ stloc.s    V_19
IL_00c8: /* 0e  | 01       */ ldarg.s    V_1
IL_00ca: /* 11  | 12       */ ldloc.s    V_18
IL_00cc: /* 11  | 13       */ ldloc.s    V_19
IL_00ce: /* 6f  | 0A00000E */ callvirt   TinyCalc.Cell get_Item(Int32, Int32)/TinyCalc.Sheet
IL_00d3: /* 13  | 10       */ stloc.s    V_16
IL_00d5: /* 11  | 10       */ ldloc.s    V_16
IL_00d7: /* 2c  | 0F       */ brfalse.s  IL_00e8
IL_00d9: /* 11  | 10       */ ldloc.s    V_16
IL_00db: /* 0e  | 01       */ ldarg.s    V_1
IL_00dd: /* 11  | 12       */ ldloc.s    V_18
IL_00df: /* 11  | 13       */ ldloc.s    V_19
IL_00e1: /* 6f  | 0A00000F */ callvirt   TinyCalc.Value Eval(TinyCalc.Sheet, Int32, Int32)/TinyCalc.Cell
IL_00e6: /* 2b  | 01       */ br.s       IL_00e9
IL_00e8: /* 14  |          */ ldnull
IL_00e9: /* 13  | 11       */ stloc.s    V_17
IL_00eb: /* 11  | 11       */ ldloc.s    V_17
IL_00ed: /* 75  | 02000010 */ isinst     TinyCalc.NumberValue
IL_00f2: /* 0b  |          */ stloc.1
IL_00f3: /* 06  |          */ ldloc.0
IL_00f4: /* 39  | 0000001A */ brfalse    IL_0113
IL_00f9: /* 07  |          */ ldloc.1
IL_00fa: /* 39  | 00000014 */ brfalse    IL_0113
```

```
IL_00ff: /* 06  |          */ ldloc.0
IL_0100: /* 7b  | 04000011 */ ldfld      Double value/TinyCalc.NumberValue
IL_0105: /* 07  |          */ ldloc.1
IL_0106: /* 7b  | 04000012 */ ldfld      Double value/TinyCalc.NumberValue
IL_010b: /* 58  |          */ add
IL_010c: /* 73  | 06000013 */ newobj     Void .ctor(Double)/TinyCalc.NumberValue
IL_0111: /* 2b  | 0A       */ br.s       IL_011d
IL_0113: /* 72  | 70000014 */ ldstr      "ARGTYPE"
IL_0118: /* 73  | 06000015 */ newobj     Void .ctor(System.String)/TinyCalc.ErrorValue
IL_011d: /* 2a  |          */ ret
```

A.5.9 Level 6 for A1=A2+A3+A4

```
IL_0000: /* 20  | 00000000 */ ldc.i4     0
IL_0005: /* 0e  | 02       */ ldarg.s    V_2
IL_0007: /* 58  |          */ add
IL_0008: /* 13  | 06       */ stloc.s    V_6
IL_000a: /* 20  | 00000001 */ ldc.i4     1
IL_000f: /* 0e  | 03       */ ldarg.s    V_3
IL_0011: /* 58  |          */ add
IL_0012: /* 13  | 07       */ stloc.s    V_7
IL_0014: /* 0e  | 01       */ ldarg.s    V_1
IL_0016: /* 11  | 06       */ ldloc.s    V_6
IL_0018: /* 11  | 07       */ ldloc.s    V_7
IL_001a: /* 6f  | 0A000002 */ callvirt   TinyCalc.Cell get_Item(Int32, Int32)/TinyCalc.Sheet
IL_001f: /* 13  | 04       */ stloc.s    V_4
IL_0021: /* 11  | 04       */ ldloc.s    V_4
IL_0023: /* 2c  | 0F       */ brfalse.s  IL_0034
IL_0025: /* 11  | 04       */ ldloc.s    V_4
IL_0027: /* 0e  | 01       */ ldarg.s    V_1
IL_0029: /* 11  | 06       */ ldloc.s    V_6
IL_002b: /* 11  | 07       */ ldloc.s    V_7
IL_002d: /* 6f  | 0A000003 */ callvirt   TinyCalc.Value Eval(TinyCalc.Sheet, Int32, Int32)/TinyCalc.Cell
IL_0032: /* 2b  | 01       */ br.s       IL_0035
IL_0034: /* 14  |          */ ldnull
IL_0035: /* 13  | 05       */ stloc.s    V_5
IL_0037: /* 11  | 05       */ ldloc.s    V_5
IL_0039: /* 75  | 02000004 */ isinst     TinyCalc.NumberValue
IL_003e: /* 0c  |          */ stloc.2
IL_003f: /* 08  |          */ ldloc.2
IL_0040: /* 39  | 0000005A */ brfalse    IL_009f
IL_0045: /* 08  |          */ ldloc.2
IL_0046: /* 7b  | 04000005 */ ldfld      Double value/TinyCalc.NumberValue
IL_004b: /* 20  | 00000000 */ ldc.i4     0
IL_0050: /* 0e  | 02       */ ldarg.s    V_2
IL_0052: /* 58  |          */ add
IL_0053: /* 13  | 0A       */ stloc.s    V_10
IL_0055: /* 20  | 00000002 */ ldc.i4     2
IL_005a: /* 0e  | 03       */ ldarg.s    V_3
IL_005c: /* 58  |          */ add
IL_005d: /* 13  | 0B       */ stloc.s    V_11
IL_005f: /* 0e  | 01       */ ldarg.s    V_1
```

```
IL_0061: /* 11  | 0A       */ ldloc.s    V_10
IL_0063: /* 11  | 0B       */ ldloc.s    V_11
IL_0065: /* 6f  | 0A000006 */ callvirt   TinyCalc.Cell get_Item(Int32, Int32)/TinyCalc.Sheet
IL_006a: /* 13  | 08       */ stloc.s    V_8
IL_006c: /* 11  | 08       */ ldloc.s    V_8
IL_006e: /* 2c  | 0F       */ brfalse.s  IL_007f
IL_0070: /* 11  | 08       */ ldloc.s    V_8
IL_0072: /* 0e  | 01       */ ldarg.s    V_1
IL_0074: /* 11  | 0A       */ ldloc.s    V_10
IL_0076: /* 11  | 0B       */ ldloc.s    V_11
IL_0078: /* 6f  | 0A000007 */ callvirt   TinyCalc.Value Eval(TinyCalc.Sheet, Int32, Int32)/TinyCalc.Cell
IL_007d: /* 2b  | 01       */ br.s       IL_0080
IL_007f: /* 14  |          */ ldnull
IL_0080: /* 13  | 09       */ stloc.s    V_9
IL_0082: /* 11  | 09       */ ldloc.s    V_9
IL_0084: /* 75  | 02000008 */ isinst     TinyCalc.NumberValue
IL_0089: /* 0d  |          */ stloc.3
IL_008a: /* 09  |          */ ldloc.3
IL_008b: /* 39  | 0000000E */ brfalse    IL_009e
IL_0090: /* 09  |          */ ldloc.3
IL_0091: /* 7b  | 04000009 */ ldfld      Double value/TinyCalc.NumberValue
IL_0096: /* 58  |          */ add
IL_0097: /* 73  | 0600000A */ newobj     Void .ctor(Double)/TinyCalc.NumberValue
IL_009c: /* 2b  | 0B       */ br.s       IL_00a9
IL_009e: /* 26  |          */ pop
IL_009f: /* 72  | 7000000B */ ldstr      "ARGTYPE"
IL_00a4: /* 73  | 0600000C */ newobj     Void .ctor(System.String)/TinyCalc.ErrorValue
IL_00a9: /* 75  | 0200000D */ isinst     TinyCalc.NumberValue
IL_00ae: /* 0a  |          */ stloc.0
IL_00af: /* 06  |          */ ldloc.0
IL_00b0: /* 39  | 0000005A */ brfalse    IL_010f
IL_00b5: /* 06  |          */ ldloc.0
IL_00b6: /* 7b  | 0400000E */ ldfld      Double value/TinyCalc.NumberValue
IL_00bb: /* 20  | 00000000 */ ldc.i4     0
IL_00c0: /* 0e  | 02       */ ldarg.s    V_2
IL_00c2: /* 58  |          */ add
IL_00c3: /* 13  | 0E       */ stloc.s    V_14
IL_00c5: /* 20  | 00000003 */ ldc.i4     3
IL_00ca: /* 0e  | 03       */ ldarg.s    V_3
IL_00cc: /* 58  |          */ add
IL_00cd: /* 13  | 0F       */ stloc.s    V_15
IL_00cf: /* 0e  | 01       */ ldarg.s    V_1
IL_00d1: /* 11  | 0E       */ ldloc.s    V_14
IL_00d3: /* 11  | 0F       */ ldloc.s    V_15
IL_00d5: /* 6f  | 0A00000F */ callvirt   TinyCalc.Cell get_Item(Int32, Int32)/TinyCalc.Sheet
IL_00da: /* 13  | 0C       */ stloc.s    V_12
IL_00dc: /* 11  | 0C       */ ldloc.s    V_12
IL_00de: /* 2c  | 0F       */ brfalse.s  IL_00ef
IL_00e0: /* 11  | 0C       */ ldloc.s    V_12
IL_00e2: /* 0e  | 01       */ ldarg.s    V_1
IL_00e4: /* 11  | 0E       */ ldloc.s    V_14
IL_00e6: /* 11  | 0F       */ ldloc.s    V_15
```

```
IL_00e8: /* 6f  | 0A000010 */ callvirt   TinyCalc.Value Eval(TinyCalc.Sheet, Int32, Int32)/TinyCalc.Cell
IL_00ed: /* 2b  | 01       */ br.s       IL_00f0
IL_00ef: /* 14  |          */ ldnull
IL_00f0: /* 13  | 0D       */ stloc.s    V_13
IL_00f2: /* 11  | 0D       */ ldloc.s    V_13
IL_00f4: /* 75  | 02000011 */ isinst     TinyCalc.NumberValue
IL_00f9: /* 0b  |          */ stloc.1
IL_00fa: /* 07  |          */ ldloc.1
IL_00fb: /* 39  | 0000000E */ brfalse    IL_010e
IL_0100: /* 07  |          */ ldloc.1
IL_0101: /* 7b  | 04000012 */ ldfld      Double value/TinyCalc.NumberValue
IL_0106: /* 58  |          */ add
IL_0107: /* 73  | 06000013 */ newobj     Void .ctor(Double)/TinyCalc.NumberValue
IL_010c: /* 2b  | 0B       */ br.s       IL_0119
IL_010e: /* 26  |          */ pop
IL_010f: /* 72  | 70000014 */ ldstr      "ARGTYPE"
IL_0114: /* 73  | 06000015 */ newobj     Void .ctor(System.String)/TinyCalc.ErrorValue
IL_0119: /* 2a  |          */ ret
```

A.5.10 Level 7 for A1=A2+A3+A4

```
IL_0000: /* 20  | 00000000 */ ldc.i4     0
IL_0005: /* 0e  | 02       */ ldarg.s    V_2
IL_0007: /* 58  |          */ add
IL_0008: /* 13  | 0A       */ stloc.s    V_10
IL_000a: /* 20  | 00000001 */ ldc.i4     1
IL_000f: /* 0e  | 03       */ ldarg.s    V_3
IL_0011: /* 58  |          */ add
IL_0012: /* 13  | 0B       */ stloc.s    V_11
IL_0014: /* 0e  | 01       */ ldarg.s    V_1
IL_0016: /* 11  | 0A       */ ldloc.s    V_10
IL_0018: /* 11  | 0B       */ ldloc.s    V_11
IL_001a: /* 6f  | 0A000002 */ callvirt   TinyCalc.Cell get_Item(Int32, Int32)/TinyCalc.Sheet
IL_001f: /* 13  | 08       */ stloc.s    V_8
IL_0021: /* 11  | 08       */ ldloc.s    V_8
IL_0023: /* 2c  | 0F       */ brfalse.s  IL_0034
IL_0025: /* 11  | 08       */ ldloc.s    V_8
IL_0027: /* 0e  | 01       */ ldarg.s    V_1
IL_0029: /* 11  | 0A       */ ldloc.s    V_10
IL_002b: /* 11  | 0B       */ ldloc.s    V_11
IL_002d: /* 6f  | 0A000003 */ callvirt   TinyCalc.Value Eval(TinyCalc.Sheet, Int32, Int32)/TinyCalc.Cell
IL_0032: /* 2b  | 01       */ br.s       IL_0035
IL_0034: /* 14  |          */ ldnull
IL_0035: /* 13  | 09       */ stloc.s    V_9
IL_0037: /* 11  | 09       */ ldloc.s    V_9
IL_0039: /* 7b  | 04000004 */ ldfld      Double value/TinyCalc.NumberValue
IL_003e: /* 13  | 06       */ stloc.s    V_6
IL_0040: /* 20  | 00000000 */ ldc.i4     0
IL_0045: /* 0e  | 02       */ ldarg.s    V_2
IL_0047: /* 58  |          */ add
IL_0048: /* 13  | 0E       */ stloc.s    V_14
IL_004a: /* 20  | 00000002 */ ldc.i4     2
```

```
IL_004f: /* 0e  | 03       */ ldarg.s    V_3
IL_0051: /* 58  |          */ add
IL_0052: /* 13  | 0F       */ stloc.s    V_15
IL_0054: /* 0e  | 01       */ ldarg.s    V_1
IL_0056: /* 11  | 0E       */ ldloc.s    V_14
IL_0058: /* 11  | 0F       */ ldloc.s    V_15
IL_005a: /* 6f  | 0A000005 */ callvirt   TinyCalc.Cell get_Item(Int32, Int32)/TinyCalc.Sheet
IL_005f: /* 13  | 0C       */ stloc.s    V_12
IL_0061: /* 11  | 0C       */ ldloc.s    V_12
IL_0063: /* 2c  | 0F       */ brfalse.s  IL_0074
IL_0065: /* 11  | 0C       */ ldloc.s    V_12
IL_0067: /* 0e  | 01       */ ldarg.s    V_1
IL_0069: /* 11  | 0E       */ ldloc.s    V_14
IL_006b: /* 11  | 0F       */ ldloc.s    V_15
IL_006d: /* 6f  | 0A000006 */ callvirt   TinyCalc.Value Eval(TinyCalc.Sheet, Int32, Int32)/TinyCalc.Cell
IL_0072: /* 2b  | 01       */ br.s       IL_0075
IL_0074: /* 14  |          */ ldnull
IL_0075: /* 13  | 0D       */ stloc.s    V_13
IL_0077: /* 11  | 0D       */ ldloc.s    V_13
IL_0079: /* 7b  | 04000007 */ ldfld      Double value/TinyCalc.NumberValue
IL_007e: /* 13  | 07       */ stloc.s    V_7
IL_0080: /* 11  | 06       */ ldloc.s    V_6
IL_0082: /* 11  | 07       */ ldloc.s    V_7
IL_0084: /* 58  |          */ add
IL_0085: /* 0c  |          */ stloc.2
IL_0086: /* 20  | 00000000 */ ldc.i4     0
IL_008b: /* 0e  | 02       */ ldarg.s    V_2
IL_008d: /* 58  |          */ add
IL_008e: /* 13  | 12       */ stloc.s    V_18
IL_0090: /* 20  | 00000003 */ ldc.i4     3
IL_0095: /* 0e  | 03       */ ldarg.s    V_3
IL_0097: /* 58  |          */ add
IL_0098: /* 13  | 13       */ stloc.s    V_19
IL_009a: /* 0e  | 01       */ ldarg.s    V_1
IL_009c: /* 11  | 12       */ ldloc.s    V_18
IL_009e: /* 11  | 13       */ ldloc.s    V_19
IL_00a0: /* 6f  | 0A000008 */ callvirt   TinyCalc.Cell get_Item(Int32, Int32)/TinyCalc.Sheet
IL_00a5: /* 13  | 10       */ stloc.s    V_16
IL_00a7: /* 11  | 10       */ ldloc.s    V_16
IL_00a9: /* 2c  | 0F       */ brfalse.s  IL_00ba
IL_00ab: /* 11  | 10       */ ldloc.s    V_16
IL_00ad: /* 0e  | 01       */ ldarg.s    V_1
IL_00af: /* 11  | 12       */ ldloc.s    V_18
IL_00b1: /* 11  | 13       */ ldloc.s    V_19
IL_00b3: /* 6f  | 0A000009 */ callvirt   TinyCalc.Value Eval(TinyCalc.Sheet, Int32, Int32)/TinyCalc.Cell
IL_00b8: /* 2b  | 01       */ br.s       IL_00bb
IL_00ba: /* 14  |          */ ldnull
IL_00bb: /* 13  | 11       */ stloc.s    V_17
IL_00bd: /* 11  | 11       */ ldloc.s    V_17
IL_00bf: /* 7b  | 0400000A */ ldfld      Double value/TinyCalc.NumberValue
IL_00c4: /* 0d  |          */ stloc.3
IL_00c5: /* 08  |          */ ldloc.2
```

```
IL_00c6: /* 09  |          */ ldloc.3
IL_00c7: /* 58  |          */ add
IL_00c8: /* 73  | 0600000B */ newobj     Void .ctor(Double)/TinyCalc.NumberValue
IL_00cd: /* 2a  |          */ ret
```

A.6 TinyScript - API and examples

A.6.1 API, classes and methods

Classes	
Workbook	Class that represents workbooks.
Sheet	Class that represents a sheet.
WorkbookIO	Class that can perform IO on a WorkBook.
FormatOptions	Class that controls the way formulas are shown.
GeneratorOptions	Class that controls how recalculations are performed.

WorkbookIO methods	
`Workbook Read(String filename)`	Reads the spreadsheet referenced by `filename` if possible. Returns the corresponding Workbook if possible. Returns `Null` if it can not be done. Spreadsheet formats accepted can be found in section 4.6.
`Boolean Write(Workbook wb, String filename)`	Writes the spreadsheet `wb` to `filename`. The format is deduced by the filename extension if possible. Otherwise the default format XMLSS is used. Returns true upon success. False upon failure.

Editing spreadsheets	
`Cell Sheet.CutCell(int col, int row`	Cuts the cell identified by `col` and `row` out of the sheet.
`Cell Sheet.CopyCell(int col, int row`	Copies the cell identified by `col` and `row`.
`Sheet.DelCell(int col, int row`	removes the cell identified by `col` and `row` if any exists.
`Sheet.PasteCell(Cell cell, int col, int row`	Inserts `cell` at `col` and `row` in the sheet.

Constructing spreadsheets	
`Workbook wb = new Workbook()`	Creates an empty Workbook.
`Sheet sheet = new Sheet(Workbook wb, String sheetname, int cols, int rows)`	Creates an empty Sheet called `sheetname` in the workbook `wb` containing `cols` columns and `rows` rows.
`Sheet sheet = new Sheet(Workbook wb, int cols, int rows)`	Creates an empty Sheet and names it using deduction on the already existing sheets. If none exists the default name "Sheet1" is used.
`Sheet Workbook[String sheetname]`	Returns the Sheet called `sheetname` if it exists. Otherwise it throws and exception of type `SheetName`.
`Sheet.AddCell(String formula, int col, int row)`	Inserts a `formula` in the `sheet` at column `col` and `row` row if a valid formula can be parsed from `formula`. Otherwise it throws an exception of type `Exception`.

Recalculations	
`GeneratorOptions`	Class that controls how code is generated/evaluated, see section for futher details.
`GeneratorOptions.level`	Method that sets the optimization level of the generated/evaluated code.
`Workbook.Recompute()`	The `Recompute` Method recomputes the current workbook at the level required by the global GeneratorOption class.

A.6.2 First Script and its output

Example 16 First test script displaying how TinyScript is used

```
using System;
using System.Collections.Generic;
using System.Text;

namespace TinyScript {
   static class program {
      static void Main(String[] args) {
         TinyCalc.WorkBookIO workbookio = new TinyCalc.WorkBookIO();
         TinyCalc.Workbook wb = new TinyCalc.Workbook();
         TinyCalc.Sheet sheet1 = new TinyCalc.Sheet(wb, "Sheet1", 7, 7);
         TinyCalc.Sheet sheet2 = new TinyCalc.Sheet(wb, "Sheet2", 7, 7);

         // FormatOptions are used to control how things are displayed in ShowXXX-Methods.
         TinyCalc.FormatOptions fo = new TinyCalc.FormatOptions();

         sheet1.AddCell("=5",0,0); //A1
         sheet1.AddCell("=7",0,1); //A2
         sheet2.AddCell("8", 0, 1); //A2
         sheet1.AddCell("=A1+Sheet2!A$2", 0, 2); //A3 = 5+8 = 13
         TinyCalc.Cell cell = sheet1.CopyCell(0,2); //A3 = A1+Sheet2!A2
         sheet1.PasteCell(cell, 0,3); //A4 = A2+Sheet2!A2
         sheet1.PasteCell(cell, 0,4); //A5
         sheet1.PasteCell(cell, 0,5); //A6
         sheet1.PasteCell(cell, 0,6); //A7

         wb.Recompute();
         Console.WriteLine("Value of Sheet1.A1: {0}", sheet1.ShowValue(0, 0));
         Console.WriteLine("Value of Sheet1.A2: {0}", sheet1.ShowValue(0, 1));
         Console.WriteLine("Value of Sheet2.A2: {0}", sheet2.ShowValue(0, 1));
         Console.WriteLine("Value of Sheet1.{0}: {1}", sheet1.Show(0, 2, fo), sheet1.ShowValue(0, 2));
         Console.WriteLine("Value of Sheet1.{0}: {1}", sheet1.Show(0, 5, fo), sheet1.ShowValue(0, 5));
      }
   }
}
```

Example 17 The output of running the first testscript with the command: `TinyCalc -f FirstTestScript.cs`

```
Script built.
Main method found (with arguments).
Running script.

No arguments given, providing dummy
Value of Sheet1.A1: 5
Value of Sheet1.A2: 7
Value of Sheet2.A2: 8
Value of Sheet1.=A1+Sheet2!A2: 13
Value of Sheet1.=A5+Sheet2!A6: #ERR: ARGTYPE
Script terminated
```

A.6.3 Second Script and its output

Example 18 Second test script displaying how TinyScript is used

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Diagnostics; // Stopwatch

namespace TinyScript {
   static class program {
      static void Main(String[] args) {
         TinyCalc.WorkBookIO workbookio = new TinyCalc.WorkBookIO();
         TinyCalc.Workbook wb1 = new TinyCalc.Workbook();
         TinyCalc.Sheet sheet1 = new TinyCalc.Sheet(wb1, "Sheet1", 7, 7);
         TinyCalc.Sheet sheet2 = new TinyCalc.Sheet(wb1, "Sheet2", 7, 7);

         // FormatOptions are used to control how things are displayed in ShowXXX-Methods.
         TinyCalc.FormatOptions fo = new TinyCalc.FormatOptions();

         sheet1.AddCell("=5",0,0);
         sheet1.AddCell("=7",0,1);
         sheet2.AddCell("8", 0, 1);
         sheet1.AddCell("=A1+Sheet2!A2", 0, 2);
         TinyCalc.Cell cell = sheet1.CopyCell(0,2);
         sheet1.PasteCell(cell, 0,3);
         sheet1.PasteCell(cell, 0,4);
         sheet1.PasteCell(cell, 0,5);
         sheet1.PasteCell(cell, 0,6);

         Boolean saved = workbookio.Write(wb1, "c:/tmptest.xml");
         if(!saved) {
            Console.WriteLine("Could not save workbook");

         } else {
            TinyCalc.Workbook wb2 = workbookio.Read("c:/tmptest.xml");

            go.level = TinyCalc.GeneratorLevel.Level0;

            wb1.Recompute(go);

            Stopwatch watch = new Stopwatch();
            watch.Reset();
            watch.Start();

            go.level = TinyCalc.GeneratorLevel.Level1;
            wb2.Recompute(go);

            watch.Stop();
            Console.WriteLine("Took {0} ms to Recompute at Level1", watch.ElapsedMilliseconds.ToString());

            Console.WriteLine("Value of wb1.Sheet1.A1: {0}", sheet1.ShowValue(0, 0));
            Console.WriteLine("Value of wb1.Sheet1.A2: {0}", sheet1.ShowValue(0, 1));
            Console.WriteLine("Value of wb1.Sheet2.A2: {0}", sheet2.ShowValue(0, 1));
            Console.WriteLine("Value of wb1.Sheet1.A3: {0}: {1}", sheet1.Show(0, 2, fo),
                              sheet1.ShowValue(0, 2));
            Console.WriteLine("Value of wb1.Sheet1.A7: {0}: {1}", sheet1.Show(0, 6, fo),
                              sheet1.ShowValue(0, 6));

            TinyCalc.Sheet wb2sheet1 = wb2["Sheet1"];
            TinyCalc.Sheet wb2sheet2 = wb2["Sheet2"];

            Console.WriteLine("Value of wb2.Sheet1.A1: {0}", wb2sheet1.ShowValue(0, 0));
            Console.WriteLine("Value of wb2.Sheet1.A2: {0}", wb2sheet1.ShowValue(0, 1));
            Console.WriteLine("Value of wb2.Sheet2.A2: {0}", wb2sheet2.ShowValue(0, 1));
            Console.WriteLine("Value of wb2.Sheet1.A3: {0}: {1}", wb2sheet1.Show(0, 2, fo),
                              wb2sheet1.ShowValue(0, 2));
            Console.WriteLine("Value of wb2.Sheet1.A7: {0}: {1}", wb2sheet1.Show(0, 6, fo),
                              wb2sheet1.ShowValue(0, 6));

            if(wb2sheet1.ShowValue(0, 2).Equals(sheet1.ShowValue(0, 2)) &&
               wb2sheet1.ShowValue(0, 6).Equals(sheet1.ShowValue(0, 6)))
            {
               Console.WriteLine("Comparison: OK");
            } else {
               Console.WriteLine("Comparison: failed");
            }
         }
      }
   }
}
```

Example 19 The output of running the first testscript with the command: `TinyCalc -f SecondTestScript.cs`

```
Script built.
Main method found (with arguments).
Running script.

No arguments given, providing dummy
Took 43 ms to Recompute at Level1
Value of wb1.Sheet1.A1: 5
Value of wb1.Sheet1.A2: 7
Value of wb1.Sheet2.A2: 8
Value of wb1.Sheet1.A3: =A1+Sheet2!A2: 13
Value of wb1.Sheet1.A7: =A5+Sheet2!A6: #ERR: ARGTYPE
Value of wb2.Sheet1.A1: 5
Value of wb2.Sheet1.A2: 7
Value of wb2.Sheet2.A2: 8
Value of wb2.Sheet1.A3: =A1+Sheet2!A2: 13
Value of wb2.Sheet1.A7: =A5+Sheet2!A6: #ERR: ARGTYPE
Comparison: OK
Script terminated
```

A.7 Test Examples

Examples of selected regression tests can be seen below.

```
//
// Tests expressions.
//
// Follows this format (not implemented as grammar, need to learn
// what the C# String API can acomplish!):
//
// Test = Formulas '%' Results [ '%' Options]
// Formulas = Formula {'@' Formula}
// Results = Result {'@' Result}
// Options = Option {'@' Option}
//
// Formula = Cell '=' formula expression
// Result = Cell '=' Expected string value
//        | Cell '=' Exception(String)
//
// Option = "skip" '=' "Level0".."Level7"
//        | "cols" '=' Integer
//        | "rows" '=' Integer
//        | "numberofsheets" '=' Integer
//        | "startlevel" '=' "Level0".."Level7"
//
// NB All skip, cols, rows... options can be substituted with
// globalskip, globalcols, globalrows, which makes the options
// global from that point onwards.
//
// Comments can appear with // and # in this file
// Empty lines are allowed too
//
// Thomas S. Iversen, 2006.

// *************** Addition
A1=5%A1=5
A1=5@A2=6@A3=A1+A2%A3=11
A1=5+6%A1=11
A1=5@A2=6@A3=7@A4=A1+A2+A3%A4=18
A1=5+6+7%A1=18
A1=5.5%A1=5.5
A1=5.5@A2=6.6@A3=A1+A2%A3=12.1
A1=5.5+6.6%A1=12.1
A1=5.5@A2=6@A3=A1+A2%A3=11.5
A1=5.5+6%A1=11.5
A1=2+-2%A1=0
A1=2+IF()%A1=#ERR: ARGTYPE
A1=2+"Thomas"%A1=#ERR: ARGTYPE
A2=5@A3=6@A1=2+A2:A3%A1=#ERR: ARGTYPE


// *************** Multiplication
A1=5@A2=6@A3=A1*A2%A3=30
A1=5*6%A1=30
A1=5.6@A2=7.8@A3=A1*A2%A3=43.68
A1=5.6*7.8%A1=43.68
A1=5.6@A2=1@A3=A1*A2%A3=5.6
A1=5.6*1%A1=5.6
A1=5.6@A2=0@A3=A1*A2%A3=0
A1=5.6*0%A1=0
A1=2*IF()%A1=#ERR: ARGTYPE
A1=2*"Thomas"%A1=#ERR: ARGTYPE
A2=5@A3=6@A1=2*A2:A3%A1=#ERR: ARGTYPE
```

```
// *************** Subtraction
A1=5@A2=6@A3=A1-A2%A3=-1
A1=6@A2=5@A3=A1-A2%A3=1
A1=-5@A2=-6@A3=A1-A2%A3=1
A1=6@A2=-5@A3=A1-A2%A3=11
A1=-6@A2=5@A3=A1-A2%A3=-11
A1=6@A2=0@A3=A1-A2%A3=6
A1=6.5@A2=5.6@A3=A1-A2%A3=0.9
A1=2--2%A1=4
A1=2-IF()%A1=#ERR: ARGTYPE
A1=2-"Thomas"%A1=#ERR: ARGTYPE
A2=5@A3=6@A1=2-A2:A3%A1=#ERR: ARGTYPE


// *************** Division
A1=5@A2=1@A3=A1/A2%A3=5
A1=5@A2=5@A3=A1/A2%A3=1
A1=5@A2=0@A3=A1/A2%A3=Infinity
A1=5@A2=0.1@A3=A1/A2%A3=50
A1=5@A2=10@A3=A1/A2%A3=0.5
A1=5.5@A2=5.5@A3=A1/A2%A3=1
A1=5.5@A2=55@A3=A1/A2%A3=0.1
A1=5.5@A2=1@A3=A1/A2%A3=5.5
A1=5.5@A2=-1@A3=A1/A2%A3=-5.5
A1=-5.5@A2=1@A3=A1/A2%A3=-5.5
A1=-5.5@A2=10@A3=A1/A2%A3=-0.55
A1=-5.5@A2=0.1@A3=A1/A2%A3=-55
A1=-5.5@A2=0@A3=A1/A2%A3=-Infinity
A1=2/IF()%A1=#ERR: ARGTYPE
A1=2/"Thomas"%A1=#ERR: ARGTYPE
A2=5@A3=6@A1=2/A2:A3%A1=#ERR: ARGTYPE

// *************** String concatenation
A1="Thomas"@A2="Maibritt"@A3=A1&A2%A3=ThomasMaibritt
A1="Thomas"@A2=5@A3=A1&A2%A3=#ERR: ARGTYPE%skip=Level7
A1="Thomas"@A3=A1&A2%A3=#ERR: ARGTYPE%skip=Level7
A1="Thomas"@A2="Maibritt"@A3="Mathias"@A4=A1&A2&A3%A4=ThomasMaibrittMathias
A1="Thomas"&"Maibritt"%A1=ThomasMaibritt
A1="Thomas"&5%A1=#ERR: ARGTYPE
A1=2&IF()%A1=#ERR: ARGTYPE
A2=5@A3=6@A1=2&A2:A3%A1=#ERR: ARGTYPE
```

A.8 TinyBench - API and examples

This appendix will briefly describe the XML file format used for TinyBench and show and example of how TinyBench are used from a TinyScript script. TinyBench exports a single class called **Benchmark**. This class has the following API:

Benchmark methods	
`public Benchmark(String title, String author)`	Constructs a **Benchmark** class. The benchmarks are denoted `title` and performed by `author`. Dates and hardware information are collected automatically by the **Benchmark** class.
`XmlElement AddDataSet(String name, String units)`	Adds a DataSet to the benchmark identified by `name`. Measurements are done in `units` units.
`XmlElement AddData(XmlElement xmldataset, String name)`	Adds data identified by `name` to a `DataSet`
`XmlElement AddSubData(XmlElement xmldata, String name, String units, int numberOfRuns)`	Adds subdata to data. Data are identified by `name`, measured in `units` and contains `numberOfRuns` subdataset on which an average time are computed.
`void AddRun(XmlElement xmlsubdata, int runNumber, String runData)`	Adds data from a run to a `SubData` section.
`Boolean Save(String filename)`	Saves the benchmarkdata to `filename`. Returns true on success and false on error.

An example of using the **Benchmark** class in a **TinyScript**:

Example 20 Example of using TinyBench in a TinyScript

```
Benchmark benchmark = new Benchmark("My benchmark", "Thomas S. Iversen");

XmlElement dataset = benchmark.AddDataSet("Evaluation time", "ms");
GeneratorOptions.UseFormulaSharing = true;

for (GeneratorLevel level = GeneratorLevel.Level0; level <= GeneratorLevel.Level7; level++)
{
    // Skip Level1
    if (level == GeneratorLevel.Level1)
        continue;

    GeneratorOptions.Level = level;

    XmlElement data = benchmark.AddData(dataset, level.ToString());
    XmlElement subdata1 = benchmark.AddSubData(data, "Evaluation time", "ms", runs);
    for (int run = 1; run <= runs; run++)
    {
        watch.Reset();
        watch.Start();
        wb.Recompute();
        watch.Stop();

        benchmark.AddRun(subdata1, run, watch.ElapsedMilliseconds.ToString());
    }
}

benchmark.Save("c:/mybenchmark.xml");
```

This will produce a `mybenchmark.xml` file containing (from which the XML format should be easily deductable):

Example 21 Example output of using TinyBench in a TinyScript

```
<?xml version="1.0"?>
<Root>
  <MetaData>
    <Title>Taylorexpansion of exp(0.5), no cellreferences</Title>
    <Author>Thomas S. Iversen</Author>
    <Date>10-06-2006 20:41:21</Date>
    <OperatingSystem>Microsoft Windows NT 5.0.2195 Service Pack 4</OperatingSystem>
    <MemoryInstalled Units="MB">768</MemoryInstalled>
    <CPU>Intel(R) Pentium(R) M processor 1.50GHz</CPU>
    <CurrentClockSpeed Units="MHz">600</CurrentClockSpeed>
    <MaxClockSpeed Units="MHz">1493</MaxClockSpeed>
  </MetaData>
  <DataSet Name="Evaluation time" Units="ms">
    <Data Name="Level0">
      <SubData Name="Evaluation time" Units="ms" NumberOfRuns="3">
        <Run RunNumber="1">10308</Run>
        <Run RunNumber="2">10523</Run>
        <Run RunNumber="3">10051</Run>
      </SubData>
    </Data>
    <Data Name="Level2">
      <SubData Name="Evaluation time" Units="ms" NumberOfRuns="3">
        <Run RunNumber="1">7521</Run>
        <Run RunNumber="2">7332</Run>
        <Run RunNumber="3">7809</Run>
      </SubData>
    </Data>
    <Data Name="Level3">
      <SubData Name="Evaluation time" Units="ms" NumberOfRuns="3">
        <Run RunNumber="1">5367</Run>
        <Run RunNumber="2">5729</Run>
        <Run RunNumber="3">5352</Run>
      </SubData>
    </Data>
    <Data Name="Level4">
      <SubData Name="Evaluation time" Units="ms" NumberOfRuns="3">
        <Run RunNumber="1">712</Run>
        <Run RunNumber="2">878</Run>
        <Run RunNumber="3">777</Run>
      </SubData>
    </Data>
    <Data Name="Level5">
      <SubData Name="Evaluation time" Units="ms" NumberOfRuns="3">
        <Run RunNumber="1">578</Run>
        <Run RunNumber="2">633</Run>
        <Run RunNumber="3">675</Run>
      </SubData>
    </Data>
    <Data Name="Level6">
      <SubData Name="Evaluation time" Units="ms" NumberOfRuns="3">
        <Run RunNumber="1">183</Run>
        <Run RunNumber="2">292</Run>
        <Run RunNumber="3">176</Run>
      </SubData>
    </Data>
    <Data Name="Level7">
      <SubData Name="Evaluation time" Units="ms" NumberOfRuns="3">
        <Run RunNumber="1">567</Run>
        <Run RunNumber="2">661</Run>
        <Run RunNumber="3">649</Run>
      </SubData>
    </Data>
  </DataSet>
</Root>
```

A TinyBench created `.xml` file can be turned into a `ploticus` plotfile by using the utility `TinyBench2ploticus` written for this sole purpose. A ploticus plotfile can then be turned into an actual image file containing an chart. This can ofcourse be scripted:

Example 22 Example a script, executing a Benchmark and generating a chart

```
REM Very very simple batch script to help me
REM generate benchmarks and pictures automaticly
REM
REM Thomas S. Iversen 2006-06-11

set filename=TaylorNoReferences
```

```
REM Generate new benchmark data?
if not "%1"=="redobenchmark" goto skipbenchmark
TinyCalc -f="%filename%.cs" -a="%filename%.xml"
:skipbenchmark

REM Generate ploticus plot script
TinyBench2ploticus "%filename%.xml" "hbars.ploticus.template.txt" %filename%.ploticus %filename%-3party-addons.xml

REM Generate gif picture
pl -gif %filename%.ploticus
del %filename%.gif
ren out %filename%.gif

REM Generate eps file.
pl -eps %filename%.ploticus
del %filename%.eps
ren out %filename%.eps
```

A.9 Benchmarks - An example

To save paper only one benchmark and one result file is included in the thesis. The rest can be found on the CD-ROM in the `Benchmark` directory and subdirectories.

A.9.1 Long reference chains

Example 23 The long reference chains benchmark

```
// Generate a long series of references

using System;
using System.Collections.Generic;
using System.Text;
using System.Xml;
using System.Diagnostics; // Stopwatch


namespace TinyCalc {
    static class program {
        static void Main(String[] args)
        {
            String filename = null;
            if (args != null && args.Length > 0)
            {
                if (args.Length > 1)
                {
                    Console.WriteLine("Only one argument is allowed (filename for the XML output");
                    return;
                }
                else
                {
                    filename = args[0];
                }
            }

            Stopwatch watch = new Stopwatch();

            String title = "LongReferenceChains";
            String author = "Thomas S. Iversen";

            // Number of Instances: 12288 * 2 = 24576
            int rows = 12288; int cols = 2;
            int startcol = 0;
            int runs = 3;
            GeneratorLevel startlevel = GeneratorLevel.Level0;
            GeneratorLevel endlevel = GeneratorLevel.Level7;


            Workbook wb = new Workbook();
            Sheet sheet = new Sheet(wb, cols + startcol, rows);
```

```
// A1;
sheet.AddCell("0.5", 0, 0);

// A2
sheet.AddCell("=A1*1.00001", 0, 1);

// A3..A<rows>
Cell cellA2 = sheet.CopyCell(0, 1);

for (int row = 2; row < rows; row++)
{
     sheet.PasteCell(cellA2, 0, row);
}

// B1;
sheet.AddCell("=SUM(A$1:A1)", 1, 0);

// B2 .. B<rows>
Cell cellB1 = sheet.CopyCell(1, 0);
for (int row = 1; row < rows; row++)
{
     sheet.PasteCell(cellB1, 1, row);
}


Benchmark benchmark = new Benchmark(title, author);

XmlElement dataset = benchmark.AddDataSet("Evaluation time", "ms");
GeneratorOptions.UseFormulaSharing = true;
for (GeneratorLevel level = startlevel; level <= endlevel; level++)
{
    // Skip Level1
    if (level == GeneratorLevel.Level1)
        continue;

    GeneratorOptions.Level = level;

    XmlElement data = benchmark.AddData(dataset, "TinyCalc SUM -- " + level.ToString());
    XmlElement subdata1 = benchmark.AddSubData(data, "Evaluation time", "ms", runs);
    for (int run = 1; run <= runs; run++)
    {

        watch.Reset();
        watch.Start();
        wb.Recompute();
        watch.Stop();

        benchmark.AddRun(subdata1, run, watch.ElapsedMilliseconds.ToString());
    }
    Console.WriteLine("Value: {0}", sheet.ShowValue(1, 0));
    Console.WriteLine("Value: {0}", sheet.ShowValue(0, 1));
```

```
                Console.WriteLine("Value: {0}", sheet.ShowValue(0, rows-1));
                Console.WriteLine("Value: {0}", sheet.ShowValue(1, rows-1));
            }


            if (filename != null)
            {
                benchmark.Save(filename);
            }
        }
}
}
```

Example 24 Output of the long reference chains benchmark

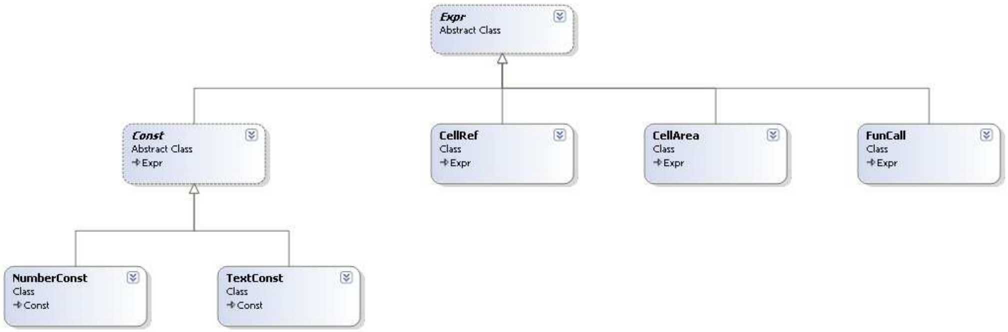
```
<?xml version="1.0"?>
<Root>
  <MetaData>
    <Title>LongReferenceChains</Title>
    <Author>Thomas S. Iversen</Author>
    <Date>10-07-2006 12:34:39</Date>
    <OperatingSystem>Microsoft Windows NT 5.0.2195 Service Pack 4</OperatingSystem>
    <MemoryInstalled Units="MB">768</MemoryInstalled>
    <CPU>Intel(R) Pentium(R) M processor 1.50GHz</CPU>
    <CurrentClockSpeed Units="MHz">600</CurrentClockSpeed>
    <MaxClockSpeed Units="MHz">1493</MaxClockSpeed>
  </MetaData>
  <DataSet Name="Evaluation time" Units="ms">
    <Data Name="TinyCalc SUM -- Level0">
      <SubData Name="Evaluation time" Units="ms" NumberOfRuns="3">
        <Run RunNumber="1">57879</Run>
        <Run RunNumber="2">59064</Run>
        <Run RunNumber="3">58950</Run>
      </SubData>
    </Data>
    <Data Name="TinyCalc SUM -- Level2">
      <SubData Name="Evaluation time" Units="ms" NumberOfRuns="3">
        <Run RunNumber="1">56773</Run>
        <Run RunNumber="2">56132</Run>
        <Run RunNumber="3">56328</Run>
      </SubData>
    </Data>
    <Data Name="TinyCalc SUM -- Level3">
      <SubData Name="Evaluation time" Units="ms" NumberOfRuns="3">
        <Run RunNumber="1">55821</Run>
        <Run RunNumber="2">55351</Run>
        <Run RunNumber="3">54732</Run>
      </SubData>
    </Data>
    <Data Name="TinyCalc SUM -- Level4">
      <SubData Name="Evaluation time" Units="ms" NumberOfRuns="3">
        <Run RunNumber="1">54754</Run>
        <Run RunNumber="2">55020</Run>
        <Run RunNumber="3">54640</Run>
      </SubData>
    </Data>
    <Data Name="TinyCalc SUM -- Level5">
      <SubData Name="Evaluation time" Units="ms" NumberOfRuns="3">
        <Run RunNumber="1">54594</Run>
        <Run RunNumber="2">54457</Run>
        <Run RunNumber="3">54997</Run>
      </SubData>
    </Data>
    <Data Name="TinyCalc SUM -- Level6">
      <SubData Name="Evaluation time" Units="ms" NumberOfRuns="3">
        <Run RunNumber="1">54642</Run>
        <Run RunNumber="2">54969</Run>
        <Run RunNumber="3">54511</Run>
      </SubData>
    </Data>
    <Data Name="TinyCalc SUM -- Level7">
      <SubData Name="Evaluation time" Units="ms" NumberOfRuns="3">
        <Run RunNumber="1">54640</Run>
        <Run RunNumber="2">54426</Run>
        <Run RunNumber="3">54922</Run>
      </SubData>
    </Data>
  </DataSet>
</Root>
```

Example 25 Third party add-on for the long reference chains benchmark

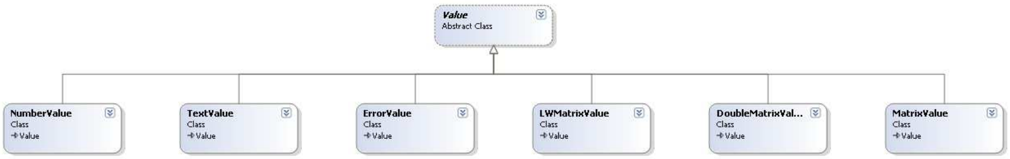
```
<?xml version="1.0"?>
<Root>
  <MetaData>
    <Title>Taylorexpansion of exp(A1), Both enumerator (A1) and denominator (factorial) are referenced</Title>
    <Author>Thomas S. Iversen</Author>
    <Date>07-06-2006 20:41:00</Date>
    <OperatingSystem>Microsoft Windows NT 5.0.2195 Service Pack 4</OperatingSystem>
    <MemoryInstalled Units="MB">768</MemoryInstalled>
    <CPU>Intel(R) Pentium(R) M processor 1.50GHz</CPU>
    <CurrentClockSpeed Units="MHz">600</CurrentClockSpeed>
    <MaxClockSpeed Units="MHz">1493</MaxClockSpeed>
  </MetaData>
  <DataSet Name="Evaluation time" Units="ms">
    <Data Name="Excel (FullCalculationRebuild)">
      <SubData Name="Evaluation time" Units="ms" NumberOfRuns="1">
        <Run RunNumber="1">323054</Run>
      </SubData>
    </Data>
    <Data Name="Excel (FullCalculation)">
      <SubData Name="Evaluation time" Units="ms" NumberOfRuns="1">
        <Run RunNumber="1">4285</Run>
      </SubData>
    </Data>
    <Data Name="Gnumeric">
      <SubData Name="Evaluation time" Units="ms" NumberOfRuns="1">
        <Run RunNumber="1">107000</Run>
      </SubData>
    </Data>
    <Data Name="OOCalc">
      <SubData Name="Evaluation time" Units="ms" NumberOfRuns="1">
        <Run RunNumber="1">17000</Run>
      </SubData>
    </Data>
  </DataSet>
</Root>
```

A.10 Class diagrams

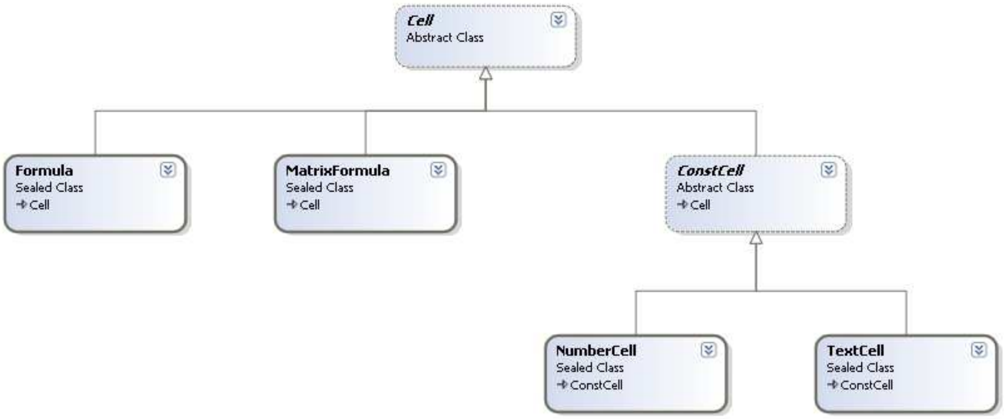
A.10.1 Classes representing Expressions



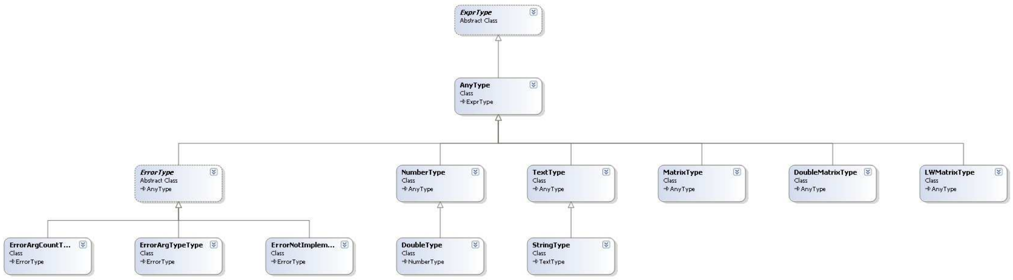
A.10.2 Classes used for Values.



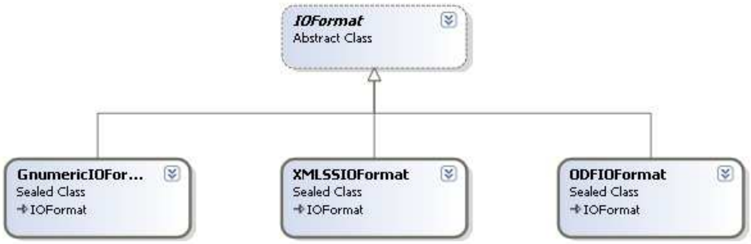
A.10.3 Classes representing Cells



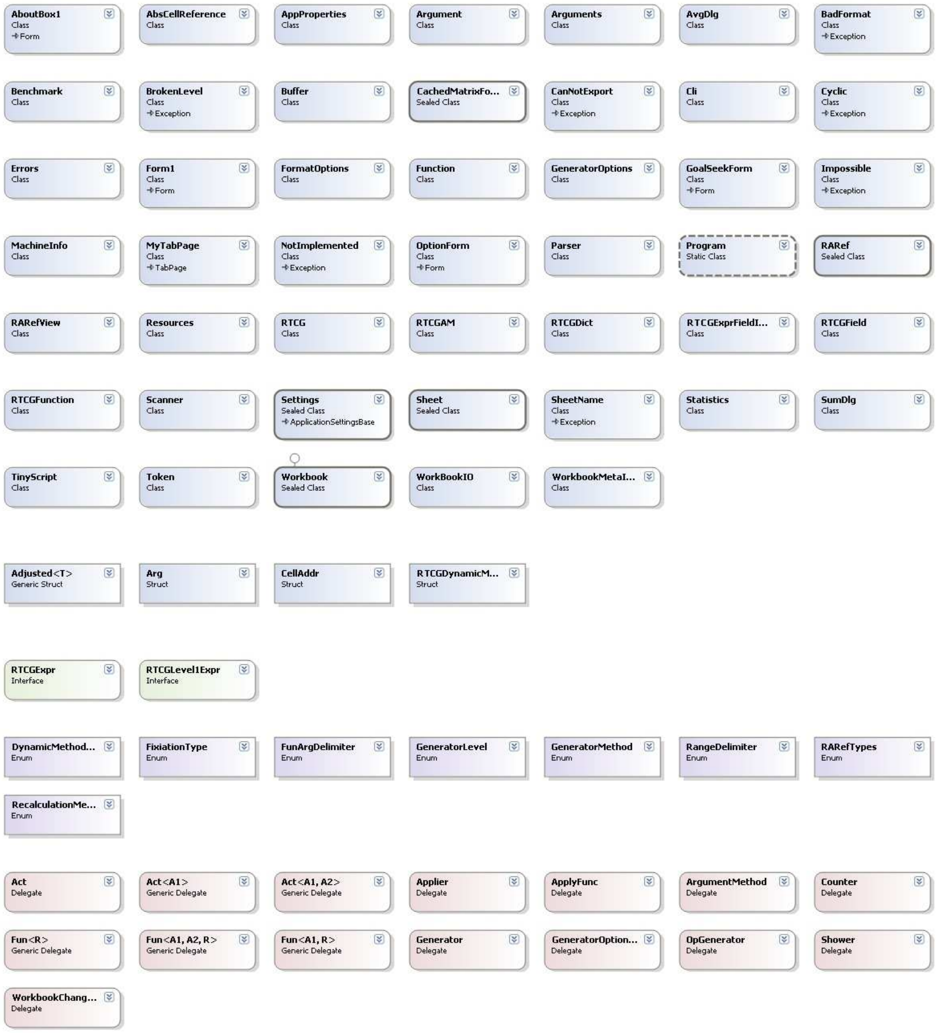
A.10.4 Classes used for type deduction



A.10.5 IOFormat classes



A.10.6 The rest



A.11 Structure of the files on the CD-ROM

The thesis is structured into three top folders: `TinyCalc`, `thesis` and `RegressionTests`. An overview over the structure of the files/directories in the `TinyCalc` directory on the CD-ROM is given below.

- `TinyCalc/` Toplevel directory containing the TinyCalc application
  - `AppProperties/AppProperties.cs` Contains code for loading and saving permanent application properties.
  - `DOM/` Document Object Model for Workbooks. Core of TinyCalc.
    - `WorkBook.cs` Implementation of workbooks
    - `Value.cs` Implementation of values
    - `Adjusted.cs`
    - `Expr.cs` Implementation of expressions.
    - `Cell.cs` Implementation of cells
    - `RARef.cs` Implementation of references.
    - `WorkbookMetaInfo.cs` Implementation of meta information regarding workbooks.
    - `Function.cs` Implementation of spreadsheet functions.
    - `Sheet.cs` Implementation of Sheets.
    - `CellAddr.cs` Implementation of methods for obtaining cell addresses.
  - `Coco/` Grammar files.
    - `Spreadsheet.ATG` COCO/R grammar for formula expressions.
    - `Parser.cs` COCO/R generated C# source for the parser.
    - `Scanner.cs` COCO/R generated C# source for the scanner.
  - `Main/Program.cs` Main entry point for TinyCalc.
  - `XML Analysis` Testsheets used for investigating the XMLSS, ODF and GNUMERIC spreadsheet formats.
  - `IO/WorkBookIO.cs` I/O Routines (XMLSS and GNUMERIC).
  - `GUI/` Implementation of the Graphical User Interface
    - `MyTabPage.cs` TabPages used to display sheets. Derived from Forms.TabPages.
    - `Form1.cs` Main form.
    - `OptionForm.cs` Form used for changing persistent options in TinyCalc.
    - `AboutBox1.cs` AboutBox displaying who made TinyCalc possible.
  - `Statistics/Statistics.cs` Class used for collecting information about a workbook. Most notably how many (possible) formulas that can be/are shared.
  - `RTCG` The generating extension implementing RTCG in TinyCalc
    - `RTCGSetup/RTCG.cs` Code for obtaining ILGenerators and collecting parameter information and setup code for the DynamicMethod and Interface Method.
    - `RTCGFunction.cs` Spreadsheet functions implemented is ILAsm with and without RTCG.
    - `RTCGAM.cs` Abstract machine used for type analysis.
    - `RTCGType.cs` Type hierarchy used for type analysis.
    - `RTCGDict.cs` Sharing of formulas is implemented in this file using an generic Dictionary.
  - `AppRuntimeOptions` Options that can be changed at runtime in TinyCalc.
    - `GeneratorOptions.cs` Controls how the generator behaves. Holds the global generatorlevel.
    - `FormatOptions.cs` Controls how formulas are shown.
    - `RARefView.cs` Controls how references are shown: A1, R1C1 or internal.
  - `TinyCalc/Types` Types used in TinyCalc.
    - `ExceptionTypes.cs` Exception types used in TinyCalc.
    - `DelegateTypes.cs` Delegate types used in TinyCalc.
    - `EnumTypes.cs` Enumeration types used in TinyCalc.
  - `CLI/` Command Line Interface for TinyCalc.
    - `Arguments.cs` Simple parsing of command line parameters.
    - `Cli.cs` Class implementing the command line interface.
  - `TinyCalc/OverheadBenchmark/Scripts` Scripts performing the overhead benchmark.
    - `RTCG_Overhead.bat` Batch file used for executing the benchmark.
    - `RTCG_Overhead.cs` C# benchmark file.
  - `TinyScript/TinyScript.cs` Implementation of TinyScript. Consists mainly of getting the script compiled and executed.
  - `BenchmarkUtils` Class used for generating XML Benchmark data.
    - `MachineInfo.cs` Obtains information on the hardware the benchmark is run.
    - `Benchmark.cs` API for doing benchmarks.
  - `Benchmarks/Scripts` Benchmarks run in this thesis. A Readme file on the actual CD-ROM will give an overview.
    - `CollectPictures.bat` Used for collecting pictures from the benchmarks.
    - `EPStothesis.bat` Copies the EPS files to the thesis/ directory
    - `dobenchmarks.bat` Performs all the benchmarks.

A.12 Source code

The source code for TinyCalc has been put on the CD-ROM in the `TinyCalc` directory. A PDF document with a “ready to print” version of the source can also be found on the CD-ROM. In the directory called `Readymade PDF files`. Furthermore it can be found on the web at:

`http://www.dina.kvl.dk/~thomassi/thesis/`