A type system for statically detecting spreadsheet errors

Ahmad, Y., Antoniu, T., Goldwater, S., & Krishnamurthi, S. (2003). A type system for statically detecting spreadsheet errors. In Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on. (pp. 174-183). doi: 10.1109/ASE.2003.1240305

A Type System for Statically Detecting Spreadsheet Errors [*]

Yanif Ahmad   Tudor Antoniu   Sharon Goldwater
Shriram Krishnamurthi
Computer Science Department, Brown University, Providence, RI 02912, USA
{*yna,taj,sgwater,sk*}*@cs.brown.edu*

Abstract

We describe a methodology for detecting user errors in spreadsheets, using the notion of units as our basic elements of checking. We define the concept of a header and discuss two types of relationships between headers, namely is-a and has-a relationships. With these, we develop a set of rules to assign units to cells in the spreadsheet. We check for errors by ensuring that every cell has a well-formed unit. We describe an implementation of the system that allows the user to check Microsoft Excel spreadsheets. We have run our system on practical examples, and even found errors in published spreadsheets.

1. Introduction

A spreadsheet is a program. That is, not only is the utility—such as VisiCalc [4] or Microsoft Excel [10]—that creates the spreadsheet a program, but so are the individual spreadsheets that we use the utility to implement. Even the humblest spreadsheet user writes simple formulas to compute results and maintain consistency between groups of data. These formulae are user programs.

Not only are spreadsheets programs, they are increasingly one of our most popular programming languages. Millions of users employ spreadsheet utilities on a regular basis. The wealth of features and tools in these utilities lets users perform several complex operations ranging from "what if" calculations to limited forms of database management. Because of their powerful operators, they are used not only in business applications [16], but in some forms of mathematical and scientific computing [14], both to teach students [7] and to build applications [15].

The widespread use of spreadsheet utilities has an unfortunate consequence: Many users have relatively little formal education in computing. Many of them learn to use spreadsheet utilities primarily by trial-and-error, by copying spreadsheets and formulae from others, and so on. Consequently, they are not trained to recognize common programming errors, and may thus fail to spot mistakes. Sociologically, it is also possible that some are more prone to trust the output of a program because it is "from the computer", not fully realizing the ways in which their actions can corrupt that output.

Unfortunately, spreadsheet utilities are often quite poor at detecting and reporting errors in user spreadsheets. There are many possible reasons for this: the desire to minimize confusing output; the expectation that spreadsheets will remain modest, not grow into large programs; and the difficulty in identifying errors and reporting them meaningfully. Sadly, this lack of checking means a great number of spreadsheets are actually buggy [11]. As spreadsheet programs grow, and business decisions and workflows increasingly rely on their output, these errors assume critical importance.

Indeed, the problems with spreadsheets are also a *software engineering* problem. Spreadsheet utilities are increasingly accessible to external programs through powerful interfaces, such as those defined for Microsoft Excel in COM [9]. This, combined with the growing desire to cobble applications from fragments in different domain-specific languages, means the reliance on spreadsheets will only grow. Therefore, the reliability of an overall software system can increasingly be compromised by a buggy spreadsheet.

In this paper, we tackle the problem of statically detecting errors in spreadsheets. We perform an operation similar to type-checking on the formulae of spreadsheets; following the lead of Erwig and Burnett [2], we call this "unit checking". We present a collection of rules that help identify weaknesses in spreadsheets that are likely to be errors.

In addition to defining rules, we have also implemented a unit checker. Building the checker and executing it on several spreadsheets helped us identify problems with prior approaches. Our unit checker operates on a mainstream spreadsheet utility, namely Microsoft Excel. By using

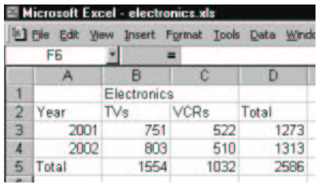
[*]Research partially supported by NSF grants ESI-0010064, ITR-0218973, SEL-0305949 and SEL-0305950.

COM, our checker interfaces directly with Excel without the need for human intervention (such as asking the user to save the contents of the spreadsheet in some other format). Excel is weaker than spreadsheet languages such as Forms/3 [1], and provides less information to build an effective checker. Nevertheless, because we do not have the power to change practice, we believe it is important to contend with the vicissitudes of a mainstream utility to make our work most widely applicable.

The rest of this paper is organized as follows. Section 2 motivates our work through a series of examples. These examples lay the groundwork for the formal material that follows. In particular, they present some of the subtleties that arise in validating spreadsheets; not treating these weakens prior work in this area. Section 3 erects a formal framework for validating spreadsheets, presenting judgement rules for units. In section 4 we briefly discuss some details of our implementation. Section 5 describes the results we obtained by testing some off-the-shelf spreadsheets. The remaining sections present related work, directions for future work, and concluding remarks.

2. Motivating Examples

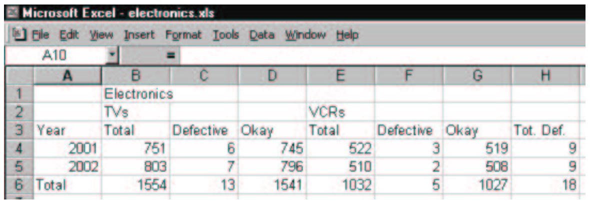
In this section we introduce the basic concepts and desired behavior of our unit checking system by providing several (intentionally simple) examples. A very simple table is shown in Figure 1. Intuitively, the user should be able to add the numbers in each row and column of the table because each row or column consists of compatible units. For example, cells B3 and B4 are both in units of TVs, so we can add them together to get another number in units of TVs. We can also add cell B3 to C3, because they both have units of the year 2001. The result, D3, will be in units of 2001, and moreover, we can abstract over the specific type of electronic device in each cell and determine that the result is also in units of Electronics. On the other hand, if the user tries to add B3 and C4 (perhaps because of a formula error), this will cause a unit-checking error, because these cells do not have either a year or a type of device in common.


Microsoft Excel - electronics.xls
File Edit View Insert Format Tools Data Windo
F6
=

	A	B	C	D
1		Electronics		
2	Year	TVs	VCRs	Total
3	2001	751	522	1273
4	2002	803	510	1313
5	Total	1554	1032	2586

Figure 1. Electronics Production by Year

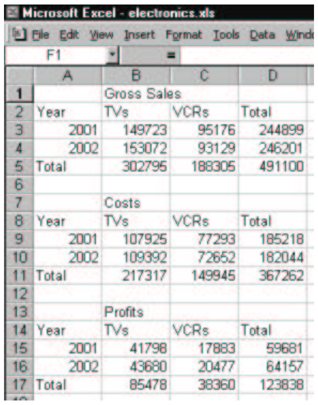
Figure 2 shows a slightly more complicated table. Here, TVs and VCRs are subdivided into three categories each.


Microsoft Excel - electronics.xls
File Edit View Insert Format Tools Data Window Help
A10
=

	A	B	C	D	E	F	G	H
1		Electronics						
2		TVs			VCRs			
3	Year	Total	Defective	Okay	Total	Defective	Okay	Tot. Def.
4	2001	751	6	745	522	3	519	9
5	2002	803	7	796	510	2	508	9
6	Total	1554	13	1541	1032	5	1027	18

Figure 2. Electronics Production Minus Defective Products

As before, we can perform an operation on cells B4 and C4 because they both have units of 2001 and abstract to units of TVs. We also want to be able to handle column H, which contains the sum of defective TVs and VCRs. We see that cells C4 and F4 have the unit 2001 in common. They are also both Electronics and Defective, but they do not have the intermediate category of either TVs or VCRs in common. Ideally, we would like our unit system to capture this information by assigning the result units of 2001 and Defective Electronics.

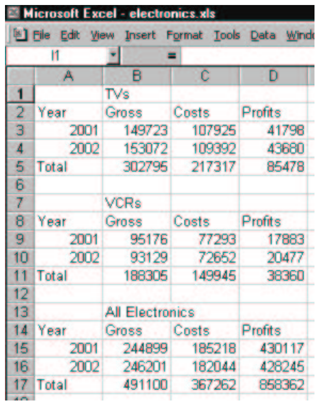

Microsoft Excel - electronics.xls
File Edit View Insert Format Tools Data Windo
F1
=

	A	B	C	D
1		Gross Sales		
2	Year	TVs	VCRs	Total
3	2001	149723	95176	244899
4	2002	153072	93129	246201
5	Total	302795	188305	491100
6				
7		Costs		
8	Year	TVs	VCRs	Total
9	2001	107925	77293	185218
10	2002	109392	72652	182044
11	Total	217317	149945	367262
12				
13		Profits		
14	Year	TVs	VCRs	Total
15	2001	41798	17883	59681
16	2002	43680	20477	64157
17	Total	85478	38360	123838

Figure 3. Electronics Sales and Profits

Finally, consider Figure 3. If we follow the pattern laid out above, D3 will have units of the year 2001 and Gross Sales, because we will "abstract" over TVs and VCRs and find that B3 and C3 have Gross Sales in common. This seems slightly odd, since Gross Sales is not a supercategory of TVs and VCRs the same way Electronics is. We also want to subtract B9 from B3 to obtain B15, but B3 and B9 have only the subcategory TVs in common, and no common supercategory at all.

In the remainder of this paper, we describe a unit system that will allow us to perform all these operations, as well as preventing errors such as adding B4 to C5 in Figure 2 (see Section 4 for the error report displayed in this case) or subtracting C9 from B3 in Figure 3. This unit system is insensitive to the specific arrangement of data, so that if the user chooses to present the data in Figure 3 differently (see Figure 4), the results of unit checking will be the same.


Microsoft Excel - electronics.xls
File Edit View Insert Format Tools Data Windo
I1
=

	A	B	C	D
1		TVs		
2	Year	Gross	Costs	Profits
3	2001	149723	107925	41798
4	2002	153072	109392	43680
5	Total	302795	217317	85478
6				
7		VCRs		
8	Year	Gross	Costs	Profits
9	2001	95176	77293	17883
10	2002	93129	72652	20477
11	Total	188305	149945	38360
12				
13		All Electronics		
14	Year	Gross	Costs	Profits
15	2001	244899	185218	430117
16	2002	246201	182044	428245
17	Total	491100	367262	858362

Figure 4. Electronics Sales and Profits: Alternative Layout

3. The Unit System

We now proceed to discussing our unit system, given the previous examples of the desired behaviour of our checker. First we describe our model of spreadsheets, defining key concepts such as headers and relationships. We then introduce units, the basic elements of our system upon which error checking occurs. In the heart of this section, we present rules to govern how units may be built from spreadsheet cells, starting with simple units, before progressing to more complex units created from cells containing mathematical operations.

3.1. Headers and Relationships

We consider spreadsheets to be comprised of cell *locations*, *values*, and *expressions*. Cell locations are given by their addresses, which we take from the Excel grid system. Values in spreadsheets are typically numbers or strings, but may include other data types as well. Some cells contain expressions, and may include operations on the values of other cells referenced by their locations. The evaluation of an expression yields a value.

A *header* is a concept that defines the common unit for a group of cells. Some cells contain values that provide names for headers, and we call these *header cells*. For example, in Figure 1, B1 is a header cell containing the value Electronics, which is the header for TVs, VCRs, and Total (in cell D2). We assume that each header cell defines a different header, unless it contains a reference to another header cell. For example, in Figure 1, the value Total appears in cell A7 as a total over Years and in cell D2 as a total over Electronics. In this case, although these header cells contain the same value, they define two different headers. On the other hand, in Figure 2, the value Defective in cell F3 comes from a reference to C3, so these two cells define the same header.

Note that a single cell may have more than one header. For example, cell B3 in Figure 1 has two headers, TVs and 2001. In addition, there may be cells whose headers are not defined explicitly by header cells. Figure 3 illustrates this situation. Here the TVs and VCRs cells are both electronic goods, so they implicitly share a header we will call Electronics, though there is no cell to indicate this. Our current solution to the problem of header inference is to rely on the user to identify the correct header units. This approach is discussed further in section 4. We assume in our unit-checking system that all headers are known.

There are two kinds of *relationships* that can exist between headers in our unit system. These relationships, common to many type systems, are the *is-a* and *has-a* relationships. We use the *is-a* relationship for both instances and subcategories, so that in Figure 1, we say that 2001 *is-a* (instance of) Year and that TVs *is-a* (subcategory of) Electronics. The *has-a* relationship generally describes properties of items or sets. For example, we can say that in Figure 3, the set of TVs *has-a* (property called) Gross Sales.

3.2. Units

Units form the basic elements upon which we perform error checking. Every cell has a unit determined by the cell's headers and the relationships those headers participate in. The simplest unit is the Top unit. Any cell that has no headers has unit Top. Examples from Figure 3 are cells A2 (Year) and B1 (Gross Sales). Header cells that participate in *is-a* relationships have hierarchical *is-a* units, which we denote with square brackets. The unit of cell C3 in Figure 2 (Defective) is therefore written Top[Electronics[TVs]]. Since all *is-a* hierarchies are ultimately derived from Top, we will generally leave Top out when describing units from this point onward.

Non-header cells have somewhat more complex units. The unit of every non-header cell contains exactly one *has-a* relationship, which we denote with braces. This is because a *has-a* relationship uniquely identifies the kind of data present in a value cell. If there were more than one *has-a* relationships, we would need to represent multiple data values in that cell, which is impossible. For the same reason we cannot have units made entirely of *is-a* relationships, although the *has-a* relationship might be implicit, as described below. In addition, non-header cells may have an arbitrary number of headers, each of which defines its own *is-a* hierarchy. We create units with multiple *is-a* hierarchies using the & operator. For example, cell B3 in Figure 3 has two headers, 2001 and TVs. The TVs header is related to the Gross Sales header by the *has-a* relationship, so the unit for B3 is:

$$\text{Electronics[TVs]\{Gross Sales\} \& Year[2001]}$$

Note that, like other headers, the header defining the *has-a* portion of a cell's unit may not be explicitly given in the spreadsheet. The tables in Figures 1 and 2, for instance, do not list this header explicitly. However, we can see by looking at the tables that the property described by the data is a Number or Quantity of electronic devices. That is, each set of devices listed in the table *has-a* Quantity. The unit for cell B3 in Figure 1 is therefore similar to the previous example:

$$\text{Electronics[TVs]\{Quantity\} \& Year[2001]}$$

Now that we have covered headers and units, we focus our attention on the description of well-formed units. We observe the following conventions for notation:

- $I(d)$ is the *is-a* header for header *d* (possibly $\emptyset$)
- $U(d)$ is the unit for header *d*
- $\mathcal{I}(a)$ is the set of *is-a* headers for the cell at location *a*
- $\mathcal{U}(a)$ is the unit for the cell at location *a*
- $d \rightarrow h$ idenitifies a *has-a* relationship between header *d* and header *h*
- $v(a)$ is the value of the cell at location *a*
- $\vec{u}(= u_1[u_2[\ldots[u_n]\ldots]])$ is the short-hand representation for a hierarchy of *is-a* relationships
- $\vec{u}[u'](= u_1[\ldots[u_n[u']]\ldots])$ is an extension of a hierarchy of *is-a* relationships, $\vec{u}$, with another *is-a* relationship, u'

The four categories of elements for which we compute units are:

1. Headers. The unit for a header is determined by its *is-a* relationships. Every header itself has either zero or one *is-a* headers. In the former case, the header's unit is Top[1]:

$$\frac{\vdash\ I(d) = \emptyset}{\vdash\ U(d) = \text{Top}}$$

   Otherwise, its unit is a concatenation of its header's unit and its header's name:

$$\frac{\vdash\ I(d) = d' \qquad \vdash\ d' \neq \emptyset \qquad \vdash\ U(d') = \vec{u}}{\vdash\ U(d) = \vec{u}[d']}$$

   We define the unit of a header cell to be the unit of the header it names.

2. Non-header cells containing values (i.e. user data), such as cell B3 in Figure 1. These cells also obtain units from their headers. Every cell containing user data must have at least one *is-a* header. Moreover, there must be exactly one *is-a* header with a *has-a* relationship. In the case where a cell has only one *is-a* header, the cell's unit is formed by concatenating its header's unit and header's name as above to obtain the *is-a* part of the unit, and adding the *has-a* header at the end:

$$\frac{\vdash \mathcal{I}(a) = \{d\} \quad \vdash \mathcal{U}(d) = \vec{u} \quad \vdash d \rightarrow h\ \ \exists!\ d}{\vdash\ \mathcal{U}(a) = \vec{u}[d]\{h\}}$$

   When a data cell has more than one *is-a* header, each *is-a* header defines its own *is-a* hierarchy, and the results are combined using 'the & operator:

$$\frac{\begin{array}{c}\vdash\ \mathcal{I}(a) = \{d, d_1, \ldots, d_n\} \\ \forall i \in 1..n : U(d_i) = \vec{u_i} \\ \vdash\ U(d) = \vec{u_d} \qquad \vdash\ d \rightarrow h\ \ \exists!\ d\end{array}}{\vdash\ \mathcal{U}(a) = \vec{u_d}[d]\{h\}\ \&\ \vec{u_1}[d_1]\ \&\ \ldots\ \&\ \vec{u_n}[d_n]}$$

3. Cells containing references only, such as cell E3 in Figure 2. The unit of a cell containing a reference is the unit of the cell it refers to.

$$\frac{\vdash\ v(a) = a'}{\vdash\ \mathcal{U}(a) = \mathcal{U}(a')}$$

4. Cells containing formulas, such as cell B5 in Figure 1. These cells contain expressions involving mathematical operators, and the resulting unit for this kind of cell depends upon the actual operator in use. We discuss the rules needed for our unit system to support the four basic mathematical operators (+, -,*,/) in the following subsection.

3.3. Units and Mathematical Operators

In this section we motivate and describe the behavior of our system with regard to mathematical operations. The

[1]The bottom part of a judgement rule is what the unit checker is able to infer based on the preconditions present in the top part of the judgement. See Pierce's book [13] for a detailed explanation of type systems.

formal judgements for these operations are listed in full in the Appendix. The section introduces these judgements in a less formal way, making use of the Excel examples.

We begin with the simplest example, Figure 1. We want to be able to add the quantity of TVs and VCRs. Intuitively, we can think of trying to union the set of TVs and VCRs to get a combined set. The resulting set will still represent quantities (the *has-a* relation) but we want the union to be described by only the common part of TVs and VCRs. In our unit notation this means that:

$$\text{Electronics[TVs]}\{\text{Quantity}\} + \text{Electronics[VCRs]}\{\text{Quantity}\}$$

when unit-checked should yield:

$$\text{Electronics}\{\text{Quantity}\}$$

Essentially, we want to keep the *has-a* part unchanged and perform a union operation, $\oplus$, on the *is-a* part of the unit. In general, we have:

$$\frac{\vdash\ \vec{u_1}\{h\} \qquad \vdash\ \vec{u_2}\{h\}}{\vdash\ \vec{u_1}\{h\} + \vec{u_2}\{h\} \rightarrow \vec{u_1} \oplus \vec{u_2}\{h\}}$$

Thus, when we add two units, if they have the same *has-a* part, the result is the union of their *is-a* part. There is an underlying principle here that is the core of the addition rule: in order to add two units together, they must have something in common (in this case the *has-a* part). Now consider the case where the two units have a common *is-a* part. Here is a variant of the example in Figure 3:

$$\text{Electronics[TVs]}\{\text{Costs}\} + \text{Electronics[TVs]}\{\text{Profits}\}$$

Clearly, we cannot perform a union operation on Costs and Profits, because they are both properties of the same set, namely TVs. By adding Costs and Profits, we obtain a new property of the same set of TVs. In general, this new property will be some irreducible combination of the two old properties. Using the $\circ$ combinator to indicate the new compound property, the result of the previous equation therefore becomes:

$$\text{Electronics[TVs]}\{\text{Cost} \circ \text{Profit}\}$$

Or, in general:

$$\frac{\vdash\ \vec{u}\{h_1\} \qquad \vdash\ \vec{u}\{h_2\}}{\vdash\ \vec{u}\{h_1\} + \vec{u}\{h_2\} \rightarrow \vec{u}\{h_1 \circ h_2\}}$$

There is only one situation that we haven't covered yet, the one where both the *is-a* part and the *has-a* part of the unit differ:

$$\text{Electronics[TVs]}\{\text{Cost}\} + \text{Electronics[VCRs]}\{\text{Profit}\}$$

This equation clearly violates our principle stating that units must have either the *is-a* or the *has-a* part in common in order for the addition to pass the checker. Intuitively, also, we see that this is the kind of operation we want to prevent, as it could only result from a mistake made by the user.

We turn our attention now to the $\oplus$ rule, as it is an important part of the addition operation. We quickly glanced over it in the first example of the section, when we obtained Electronics from Electronics[TVs] $\oplus$ Electronics[VCRs]. The $\oplus$ rule applied to the is-a parts of the units, and combined them by retaining in the result only the common parts of the two units. Judging from our first example, it might seem that the result of the union operation will always be a more general unit than either of the two arguments. But suppose we want to perform a union operation on these two units:

$$\text{Electronics[TVs[Defective]]} \oplus \text{Electronics[VCRs[Defective]]}$$

In this case we could also say that the result should be Electronics, but we would lose information common to the two original units: the fact that they are both defective. Instead, our desired result is:

$$\text{Electronics[Defective]}$$

The $\oplus$ operation therefore combines the *is-a* parts of two units creating a new unit from all the common features of the two units, not just the most general ones.

To summarize, the addition rule applies only to units that either have identical *has-a* parts, in which case the result is a $\oplus$ operation on their *is-a* parts; or identical *is-a* parts, in which case the result is a $\circ$ operation on their *has-a* parts.

Now that we have seen how addition works, we will describe subtraction. As with addition, we want to allow subtraction only between cells that have either identical *has-a* or *is-a* parts. We begin with the first case. In Figure 2, the Okay column for TVs requires us to subtract the following two units:

$$\text{Electronics[TVs[Total]]}\{\text{Quantity}\} - \text{Electronics[TVs[Defective]]}\{\text{Quantity}\}$$

We want our unit checker to identify the result as representing a quantity of TVs:

$$\text{Electronics[TVs]}\{\text{Quantity}\}$$

We cannot be any more specific about the unit of the result, since there is no way to know in general whether the set resulting from a subtraction operation contains any items of the subtracted type. In other words, we may not have subtracted all the defective TVs from the original set. We only know that, since both original sets were types of TVs, we must still have a set containing only TVs (of some type).

This result is satisfying, since it exactly mirrors the behavior of addition, where we apply $\oplus$ operator to the *is-a* parts.

Now consider subtracting two units with a common *is-a* part, as in Figure 3, where the data in the Profit column is given by:

$$\text{Electronics[TVs]}\{\text{Gross Sales}\} - \text{Electronics[TVs]}\{\text{Costs}\}$$

As with addition, the result is the combination of the *is-a* part, Electronics[TVs], and a new property derived from Gross Sales and Costs:

$$\text{Electronics[TVs]}\{\text{Gross Sales} \circ \text{Costs}\}$$

Having seen how addition and subtraction work, we can conclude that any binary operator must correctly handle two cases: identical *is-a* parts and identical *has-a* parts. In the case of identical *is-a* parts, the result of the operation is always a compound of the two different *has-a* parts. For example, in the computation for the area of a rectangle:

$$\text{Shape[Rectangle]}\{\text{Length}\} \times \text{Shape[Rectangle]}\{\text{Width}\}$$

It is obvious we want to remember that the result is given by the combination of Length and Width:

$$\text{Shape[Rectangle]}\{\text{Length} \circ \text{Width}\}$$

We conclude, therefore, that when dealing with identical *is-a* parts, any binary operator returns a $\circ$ combination of the *has-a* parts along with the *is-a* part as the result.

Is the case of identical *has-a* parts also uniform across all binary operators? We have seen that both addition and subtraction require the use of the $\oplus$ operator on the different *is-a* parts. But suppose we have the following equation:

$$\text{Shape[Rectangle]}\{\text{Length}\} \times \text{Shape[Square]}\{\text{Length}\}$$

Clearly it does not make sense to have Shape{Length} as the result. In fact, there is no satisfactory combination of the two *is-a* parts that will accurately describe the result. However, we do not want to flag this as an error, since there might be a legitimate reason for the user to perform this operation. Therefore, when dealing with any binary operator other than + or -, the result of combining two units with different *is-a* parts and the same *has-a* part is always Top$\{h\}$ (where h is the common *has-a* part).

To obtain meaningful results from constructs such as:

$$\text{Shape[Square]}\{\text{Length}\} \times \text{Shape[Square]}\{\text{Length}\}$$

the identical *is-a* combination, $\circ$, takes precedence over the identical *has-a* combination, $\oplus$ or *Top*.

Finally, we will describe the *and*(&) operation. As noted in section 2, the unit of cell B3 from Figure 1 is:

$$\text{Electronics[TVs]}\{\text{Quantity}\} \ \&\ \text{Year[2001]}$$

The unit of a value cell that has more than one header is given by the & constructor on the units inferred from each individual header. There are restrictions on the types of the units on which we can perform &.

Each header conveys a distinct property for the data in the cell, which means that a well-formed & unit consists of different, header inferred, units containing only *is-a* parts, with only one of them potentially having a *has-a* part. Since there is only one *has-a* part at most, the difference applies to the *is-a* parts of the units. Two *is-a* parts are different if and only if their top labels are different because only then do the two *is-a* parts represent disjoint data properties. The & operation is idempotent to handle the special case when two *is-a* parts are identical. For example, Electronics and Year are clearly different so it is correct to join them through &. On the other hand,

$$\text{Electronics[TVs]} \ \&\ \text{Electronics[VCRs]}\{\text{Gross Sales}\}$$

does not represent a valid & unit operation because both headers represent Electronics and that contradicts our requirement that the headers differ.

The & operation is distributive with respect to any other binary operation between units. For example, in Figure 1, cell B5 has unit:

$$\text{Electronics[TVs]}\{\text{Quantity}\} \ \&\ \text{Year[2001]} + \text{Electronics[TVs]}\{\text{Quantity}\} \ \&\ \text{Year[2002]}$$

which reduces to:

$$\text{Electronics[TVs]}\{\text{Quantity}\} \ \&\ (\text{Year[2001]} + \text{Year[2002]})$$

We want the unit checker to perform the addition on the two *is-a* units as if there were an empty *has-a* part, yielding the following result:

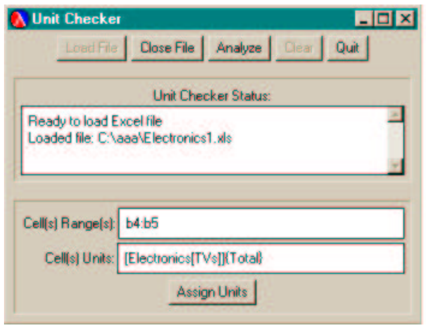
$$\text{Electronics[TVs]}\{\text{Quantity}\} \ \&\ \text{Year}$$

We thus handle the reduction of Year[2001] + Year[2002] using the special case of the identical *has-a* rule for binary operations, the one with empty *has-a* parts.

4. Implementation

Our unit checker is implemented in the DrScheme programming environment [6]. It has three components: a GUI, an I/O layer that mediates the communication with Excel, and the unit checker itself.

Figure 5 presents the GUI interface. The user can start an instance of a desired spreadsheet with the `Load File` button. `Analyze` will check the loaded spreadsheet once it is annotated with the right units. In earlier sections we assumed the existence of a header inference algorithm to correctly annotate cells with units based on header labels. Header inference is a difficult artificial intelligence and natural language processing problem. Our current implementation does not have such an algorithm. Instead, the value cells are annotated with the right units through the GUI. A range of cells is selected either with the mouse in Excel, or textually entered in the `Cell(s) Range(s)` field in the GUI, and their corresponding unit in the `Cell(s) Units` field. The `Assign Units` button updates, in Excel, the comment field of every cell from the range with the assigned unit. In the future we plan to provide a more automated process for header inference. To that end, we may do semantic analysis on headers to determine relationships with the aid of WordNet [5].



Ready to load Excel file
Loaded file: C:\aaa\Electronics1.xls

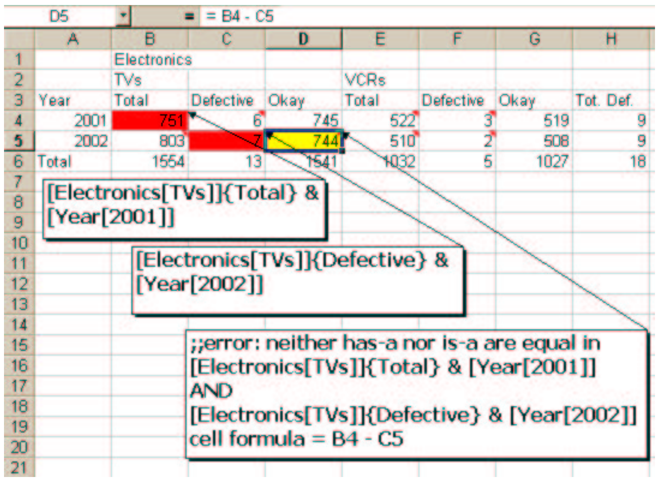
Cell(s) Range(s): b4:b5
Cell(s) Units: [Electronics[TVs]]{Total}

Figure 5. Unit Checker GUI

We use the MysterX [18] Component Object Model (COM) extension to DrScheme to communicate with Excel. Although Excel has a complex COM model (over 400 COM interfaces), we only need a few of those for our tool. Examples of the interfaces we need are: `_Application`, `_Worksheet`, `_Irange`. Each of these has methods and properties through which the application can mimic any user interaction with Excel. Our I/O layer implements the functionality needed by the tool (such as coloring cells, adding comments to cells, etc.) by using MysterX methods.

Unit errors are reported back to the user by coloring the offending cells. Figure 6 is similar to Figure 2 except we have deliberately introduced an error in cell `D5`. For each error cell, the user can display the input cells to the formula in that cell by right clicking the mouse. In Figure 6 the input cells for `D5` are `B4` and `C5`. There are two kinds of errors reported back. The unit errors are those where the unit checker identified a problem in the formula for the cell, as is the case with cell `D5`. There are also propagation errors. Those are cells that use a reference to another cell flagged by the unit checker as containing an error.

Our judgements do not dictate an order on checking a spreadsheet. In our current implementation we assign units to value cells through the GUI. All the other cells will



D5 = B4 - C5

	A	B	C	D	E	F	G	H
1		Electronics						
2		TVs			VCRs			
3	Year	Total	Defective	Okay	Total	Defective	Okay	Tot. Def.
4	2001	751	6	745	522	3	519	9
5	2002	803	7	744	510	2	508	9
6	Total	1554	13	1541	1032	5	1027	18

[Electronics[TVs]]{Total} &
[Year[2001]]

[Electronics[TVs]]{Defective} &
[Year[2002]]

;;error: neither has-a nor is-a are equal in
[Electronics[TVs]]{Total} & [Year[2001]]
AND
[Electronics[TVs]]{Defective} & [Year[2002]]
cell formula = B4 - C5

Figure 6. Unit Error

contain formulas and therefore their units will be inferred. From Excel we extract a list of all the formulas and then unit check each one in a recursive manner by computing units for subformulas first and then combining them by using our inference rules. Since Excel provides cycle detection, we do not concern ourselves with circularity and furthermore this approach is guaranteed to terminate.

Excel has over 300 pre-defined functions. These functions can be grouped together based on their domains. For example there are financial functions, matrix operators, trigonometric functions, etc. We chose representatives of each of these groups and implemented inference rules for them based on our base inference rules. In order to be fully functional, our checker would need to implement judgement rules for all the functions; this is an important step for transforming our checker from a prototype to a product.

5. Experimental Validation

Designing a type system is not hard; the difficulty is in designing one that (a) actually catches errrors, while (b) not rejecting too many meaningful programs, all while (c) running in reasonable time. The best validation of our unit checker would be to run it against off-the-shelf spreadsheets and check for all three criteria. To this end, we used the examples in a book by Filby [14] on spreadsheets for science and engineering. (None of the spreadsheets had any unit annotations. We used our tool to annotate the value cells in those spreadsheets with the correct units.) At the outset, we did not expect to find errors; we were primarily interested in whether the checker would reject any (correct) spreadsheets, and secondarily in how quickly it would run.

The table in Figure 7 describes the spreadsheets we used to test our unit checker, and the checker's performance on

Author	Description	Size	Time	COM	Checker	Error?
S. Leharne	Acid Base Titration	109	0:24	0:23	0:01	
W.J. Orvis	Oscillations Frequency	43	0:19	0:18	0:01	
	Oscillations Euler Method	345	1:52	1:51	0:01	
A.A. Gorni	Cubic Crystalline Systems X-Ray Diffraction	83	0:40	0:39	0:01	
W.J. Orvis	Electron Drift Velocity in GaAs	44	0:16	0:15	0:01	
J.P. LeRoux	Cleavage Strike Direction	236	1:16	1:13	0:03	X
	Palaeocurrent	284	1:40	1:38	0:02	
	Untilt	53	0:22	0:21	0:01	
	Chi-square	41	0:16	0:15	0:01	
A.A. Gorni	Grain size of microstructure	40	0:23	0:22	0:01	
E. Neuwirth	Feigenbaum Diagram	1000	2:58	2:57	0:01	
E. Neuwirth	Simple Model	54	0:07	0:06	0:01	
	Parametric Model	55	0:10	0:09	0:01	
	Complex Model	56	0:13	0:12	0:01	
	Complex Model with Table	75	0:19	0:18	0:01	
	Complex Model with Stepwidth	57	0:08	0:07	0:01	
	Volterra-Lotka Model	8004	14:59	14:38	0:21	
	Planets	4001	12:34	12:18	0:16	
	Planets Halfstep	4001	10:24	10:10	0:14	
W.J. Orvis	Blackbody spectral emission	507	0:53	0:52	0:01	
A.A. Gorni	Viscometric molecular weight	41	0:47	0:46	0:01	
A.A. Gorni	Point count method	26	0:18	0:17	0:01	

Figure 7. Experimental Results

them. The size given is the number of non-empty cells. The “Time” column shows the total time taken by the checker.[2] This time is largely an artifact of our use of the COM Automation interface to interact with Excel; COM Automation is known to be slow, and its overhead can be eliminated by using COM Direct Interfaces or .NET. The next two columns indicate the division of time between the COM interface and the actual checker, which clearly demonstrates that most of the time is spent interacting with Excel, not in the core of the checking procedure.

The last column, “Error?”, indicates whether our checker claimed to find an error in the corresponding spreadsheet. The table shows that it did report one purported error. To our surprise, this is a genuine error in the published spreadsheet! The author of that spreadsheet uses an Excel operator, `FREQUENCY`, that takes two ranges of cells as arguments. The second range is incorrect: one of the cells it refers to contains no data.

To run our checker, we must annotate value cells with units. This took very little time and effort, because we used the header annotations on the tables. In particular, we did not need to understand the problem domains to make these annotations. Our experience therefore suggests that the unit checker offers great promise to be a useful tool for real-world users of spreadsheets.

[2]All times are in minutes:seconds.

6. Related work

The most closely related work is the unit checker of Erwig and Burnett [2]. While based on similar principles, our systems are significantly different. Their work does not distinguish between the *is-a* and *has-a* relationships, which makes their type system much weaker and much more likely to incorrectly report errors. They ignore operators such as subtraction, which are of obvious importance. Furthermore, they do not provide an implementation or experimental validation, which we believe would have easily identified these weaknesses. (They discuss a prototype implementation in a subsequent paper [3], but it is not graphical, does not integrate cleanly with a utility like Excel, and suffers from the weaknesses of their typing rules.)

To highlight the differences in the core of the unit system, we present the following example of a reasonable spreadsheet that successfully passes our checker but would fail in that of Erwig and Burnett [2]. In Figure 4, we rearrange the tables in Figure 3, and assume that the header inference is able to infer that TVs and VCRs are both types of electronic goods. Consider the operation in cell B15. First we discuss how Erwig and Burnett’s checker would operate in this situation. In their system, cells B3 and B9 have units All Electronics[TVs[Gross]] and All Electronics[VCRs[Gross]] respectively. The subsequent addition operation in cell B15 fails, because the hierarchies of the two units differ in their second components (TVs vs. VCRs), despite the common third component of Gross. The header inference could conceivably reverse the hierarchy of the units. Cells B3 and B9 could be assigned units of Gross[TVs] and Gross[VCRs], enabling cell B15 to pass the unit checking. However the computation of profits, in cell D3 for example, would now fail (Cost[TVs] cannot be subtracted from Gross[TVs]). Our system handles this case in exactly the same manner as described above. Cell B15 turns out to be an addition of:

$$\begin{gathered}\text{All Electronics[TVs]}\{\text{Gross}\} + \\ \text{All Electronics[VCRs]}\{\text{Gross}\} = \\ \text{All Electronics}\{\text{Gross}\}\end{gathered}$$

Cell D3 is:

$$\begin{gathered}\text{All Electronics[TVs]}\{\text{Gross}\} - \\ \text{All Electronics[TVs]}\{\text{Cost}\} = \\ \text{All Electronics[TVs]}\{\text{Gross} \circ \text{Cost}\}\end{gathered}$$

This demonstrates that despite any rearrangement of the tables, providing the header inference is able to determine the relationships in the manner above, our rules may be consistently applied. Erwig and Burnett's system is unable to handle an intuitive way of tabulating data, and no rearrangement of headers is able to account for the differences in the *is-a* and *has-a* relationships.

There has been other work tackling the specific problem of detecting errors in spreadsheets. Rothermel et al. [17] apply an adaptation of testing mechanisms for imperative programs to spreadsheets. This aims at detecting the most common of spreadsheet errors, cell reference errors in cell expressions [12], through the use of data flow adequacy criteria. The authors define the data flow test adequacy criteria employed, in terms of definition-use (du) associations that are involved in visible cell outputs. Relying on user interaction to validate the values in cells, the system marks du-associations as having been exercised, and visually reflects the percentage of all du-associations exercised per cell with shades of colors. Rothermel et al. apply this kind of testing to the Forms/3 spreadsheet language [1], whereas our system pertains to Excel spreadsheets. Specifically, Excel spreadsheets are able to detect the use of blank cells in cell expressions. Thus the types of errors we are able to detect are of a different nature, and this belief is reinforced if we consider the following example. In Figure 1, suppose the cell B5 contained the cell expression B3 + C4. Our unit system would flag an error due to the addition:

$$\begin{gathered}\text{Electronics[TVs]}\{\text{Quantity}\} \;\&\; \text{Year[2001]} + \\ \text{Electronics[VCRs]}\{\text{Quantity}\} \;\&\; \text{Year[2002]}\end{gathered}$$

However the system in [17] would not be able to detect this problem in Figure 1.

Kennedy [8] describes an extension of strongly-typed programming languages to include polymorphic dimension types for values within the language, stemming from a similarity between well-typedness of programs and dimensional consistency of mathematical expressions. The author introduces the concepts of base dimensions such as length, mass, time, etc. The counter to these are derived dimensions, for example acceleration which can be thought of as length divided by time squared. Units are also discussed. These units are not to be confused with the units we check, but are units of measurement, such as meters, or kilograms. Base units are used to measure the base dimensions. Alternative units of measurement are just a scaling of base units.

7. Conclusion and Future Work

In this paper we have presented a methodology for detecting errors of a semantic nature in spreadsheets. We have introduced the concept of the *is-a* and the *has-a* relationships, whose essence is found in a large number of type systems, into the domain of spreadsheets. In the process, we have enhanced the completeness of our system in comparison to existing work, by broadening the range of units that may pass unit checking. This is important because users will disable a checker if it reports errors on valid inputs.

Our implementation provides a simple interface to unit check an Excel spreadsheet. While we have not delved deeply into the problem of header inference for a wide range of spreadsheets, our unit checker is able to detect errors in many cases without any additional information than that present in the spreadsheet, assuming header inference which we currently do through manual annotation.

There are many directions for future work. To make this system truly practical, we must complete three tasks. The first is to support the entire suite of Excel's built-in operators, so that we can handle all spreadsheets. The second is to perform a detailed study with typical users to assess the strengths and weaknesses of our interface. Finally, we have to make the overall running time of the checker much smaller so that users can integrate this tool in their development cycle. There are at least two major remaining research problems. One is building a non-trivial header inference engine, preferably one that (a) makes significant use of natural language processing techniques to maximize the accuracy of its inferences, and (b) queries users to validate its inferences and uses their feedback to adaptively improve its output. The other is to integrate Kennedy's [8] dimension checking into our system.

References

[1] M. M. Burnett, J. Atwood, R. Djang, H. Gottfried, J. Rechwein, and S.Yang. Forms/3: A first-order visual language to

explore the boundaries of the spreadsheet paradigm. *Journal of Functional Programming*, 10(2):155–206, 2001.

[2] M. Erwig and M. Burnett. Adding apples and oranges. In *Practical Aspects of Declarative Languages (PADL)*, 2002.

[3] M. Erwig and M. Burnett. Visually Customizing Inference Rules About Apples and Oranges In *2nd IEEE International Symposium on Human Centric Computer Languages and Environments*, 2002.

[4] D. Bricklin and B. Frankston. VisiCalc. http://www.bricklin.com/visicalc.htm.

[5] C. Fellbaum, editor. *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

[6] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. 12(2):159–182, 2002.

[7] A. I. Katz. Academic Computing In U.S. Colleges And Universities: A Survey. *Journal of Information Systems Education*, 4(4), December 1992.

[8] A. Kennedy. Dimension types. In D. Sannella, editor, *European Symposium on Programming*, volume 788, pages 348–362, Edinburgh, U.K., 11–13 1994. Springer.

[9] Microsoft Corporation. Microsoft Component Object Model. http://www.microsoft.com/com.

[10] Microsoft Corporation. Microsoft Excel. http://www.microsoft.com/excel.

[11] R. Panko. Finding Spreadsheet Errors: Most spreadsheet models have design flaws that may lead to long-term miscalculations. In *InformationWeek*, May 1995. http://www.informationweek.com/529/29uwfw.htm.

[12] R. Panko and R. Halverson. Spreadsheets on trial: A survey of research on spreadsheet risks. In *Twenty-Ninth Hawaii International Conference on System Sciences*, January 1996.

[13] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[14] G. Filby. *Spreadsheets in Science and Engineering*. Springer, 1995.

[15] A. Ricadela and J. Maselli. To The Middle: Big ERP vendors haven't done well in the midmarket. Can Microsoft do better?, May 2002. http://www.informationweek.com/story/IWK20020-517S0043.

[16] E. Colkin. Nasdaq Giving XBRL A Try. In *InformationWeek*, August 2002. http://www.informationweek.com/story/IWK20-020806S0004.

[17] G. Rothermel, M. Burnett, L. Li, C. Dupuis, and A. Sheretov. A methodology for testing spreadsheets. *ACM Transactions on Software Engineering and Methodology*, 10(1):110–147, 2001.

[18] P. A. Steckler. MysterX: A Scheme toolkit for building interactive applications with COM. In *Technology of Object-Oriented Languages and Systems*, pages 364–373. IEEE, August 1999.

A. Appendix: Unit inference rules

- $\nabla \equiv$ any binary operator
- $\nabla^* \equiv$ any binary operator except for +/-
- All other relations are defined in section 3.2

Unit construction rules:

Headers:

$$\frac{\vdash\ I(d) = \emptyset}{\vdash\ U(d) = \text{Top}}$$

$$\frac{\vdash\ d' \in I(d),\ U(d') = \vec{u}}{\vdash\ U(d) = \vec{u}[d']}$$

Values:

$$\frac{\vdash\ d \in \mathcal{I}(a) \qquad \vdash\ U(d) = \vec{u} \qquad d \rightarrow h}{\vdash\ \mathcal{U}(a) = \vec{u}[d]\{h\}}$$

$$\frac{\begin{array}{c}\vdash\ \{d, d_1, \ldots, d_n\} = \mathcal{I}(a)\ ,\ \forall i \in 1..n : U(d_i) = \vec{u_i} \\ U(d) = \vec{u_d}\ ,\ d \rightarrow h\end{array}}{\vdash\ \mathcal{U}(a) = \vec{u_d}[d]\{h\}\ \&\ \vec{u_1}[d_1]\ \&\ \ldots\ \&\ \vec{u_n}[d_n]}$$

References:

$$\frac{\vdash\ v(a) = a'}{\vdash\ \mathcal{U}(a) = \mathcal{U}(a')}$$

$\oplus$-rule :

$$\vec{u_1} = c_1[\ldots[c_i[x_1\ldots[x_k[c_{i+1}\ldots[c_j]\ldots]]\ldots]]\ldots]$$

$$\vec{u_2} = c_1[\ldots[c_i[y_1\ldots[y_l[c_{i+1}\ldots[c_j]\ldots]]\ldots]]\ldots]$$

$$\frac{\vdash \vec{u_1} \qquad \vdash \vec{u_2} \qquad i > 0;\ j \geq i;\ k, l \geq 0}{\vdash \vec{u_1} \oplus \vec{u_2} \rightarrow c_1[\ldots[c_i\ldots[c_j]\ldots]\ldots]}$$

&-rule:

$$\vec{u} = u_1[\ldots[u_i]\ldots]$$

$$\vec{v} = v_1[\ldots[v_j]\ldots]$$

$$\frac{\vdash \vec{u} \quad \vdash \vec{v} \quad u_1 \neq v_1}{\vdash\ \vec{u}\,\&\,\vec{v}}$$

Simplification rules:

$$\vec{u_1}\ \&\ (\vec{u_2}\ \&\ \vec{u_3}) = (\vec{u_1}\ \&\ \vec{u_2})\ \&\ \vec{u_3} = \vec{u_1}\ \&\ \vec{u_2}\ \&\ \vec{u_3}$$

$$\vec{u_1}\ \&\ \vec{u_2} = \vec{u_2}\ \&\ \vec{u_1}$$

$$\vec{u}\ \&\ \vec{u} = \vec{u}$$

$$\vec{u_1}\ \&\ (\vec{u_2}\ \nabla\ \vec{u_3}) = \vec{u_1} \& \vec{u_2}\ \nabla\ \vec{u_1} \& \vec{u_3}$$

Identical *is-a* rule:

$$\frac{\vdash\ \ \vec{u}\{h_1\} \qquad \vdash\ \ \vec{u}\{h_2\}}{\vdash\ \ \vec{u}\{h_1\}\nabla\vec{u}\{h_2\} \rightarrow \vec{u}\{h_1 \circ h_2\}}$$

Identical *has-a* rule, *has-a* can be empty:

$$\frac{\vdash\ \ \vec{u_1}\{h\} \qquad \vdash\ \ \vec{u_2}\{h\}}{\vdash\ \ \vec{u_1}\{h\} \pm \vec{u_2}\{h\} \rightarrow \vec{u_1} \oplus \vec{u_2}\{h\}}$$

$$\frac{\vdash\ \ \vec{u_1}\{h\} \qquad \vdash\ \ \vec{u_2}\{h\}}{\vdash\ \ \vec{u_1}\{h\}\nabla^*\vec{u_2}\{h\} \rightarrow Top\{h\}}$$