

A Framework for Checking Spreadsheets

Tudor Antoniu - taj@cs.brown.edu

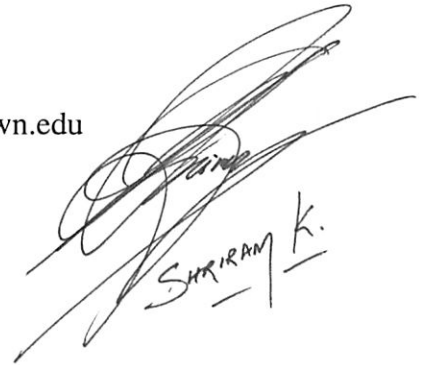
Advisor: Shriram Krishnamurthi - sk@cs.brown.edu

Department of Computer Science

Brown University

Providence, RI 02912

May 7, 2003

A handwritten signature in black ink, appearing to be 'Shriram K.', with a large, stylized flourish above it.

Abstract

We have created a framework for detecting user errors in spreadsheets. It consists of a unit-checker (XeLda) and a type-checker. The unit-checker enforces unit consistency and flags misused units and derived units that clash with their annotations. The type checker assigns types to cells based on their header labels. There are two types of relationships between headers, namely *is-a* and *has-a* relationships. We develop a set of rules to assign types to cells. We check that every cell has a well-formed type. Our framework is sensitive to the intricacies of Excel spreadsheets, and can handle tables, matrices, and even circular references. Our approach draws on the idea of *unit inference* for programming languages developed by Goubault, Kennedy, Wand, and others. The framework integrates smoothly with Excel by accessing its COM interfaces from PLT Scheme. Our technology can detect errors in the consistency of some off-the-shelf scientific spreadsheets.

Contents

1	Introduction	4
2	Types	6
2.1	Motivating Examples	6
2.2	Headers and Relationships	7
2.3	Types	10
2.4	Types and Mathematical Operators	13
3	Units	17
3.1	A XeLda Tutorial	17
3.2	Deriving Units	20
3.3	Tables and Matrices	23
3.4	Circular References	26
3.4.1	Constraint Generation	26
3.4.2	Constraint Resolution	28
4	Implementation	32
4.1	Excel, COM, and MysterX	32
4.2	Mostly-Functional COM Scripting	33
5	Applying the Checkers	35
6	Related Work	38
7	From Prototype to Product	41

Chapter 1

Introduction

The formula languages used to program spreadsheet applications are today's most widely-used functional languages. While such languages lack many of the sophisticated features and abstraction mechanisms found in languages such as Scheme, ML, and Haskell, their core is an effect-free language of expressions. Programming language researchers would do well to pay some attention to such languages, because research results for them can have a big impact. To date, only a modest amount of language research has been directed toward spreadsheets.

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 [31], but in some forms of mathematical and scientific computing, both to teach students [33] and to build applications [32].

Spreadsheet applications appeal to many users because their formula languages are expressive enough to create complex computational models without in-depth programming knowledge. But spreadsheet programs are particularly likely to contain errors: one reason is that users are likely not to be professional programmers, who may not use rigorous software design and testing methods [15]. Given the large numbers of spreadsheet users, and the likelihood of errors, there is a great potential payoff for automatic detection of spreadsheet errors.

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 [34]. 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.

Spreadsheet formulas are commonly used to perform financial, scientific, and engineering calculations, which manipulate quantities with associated units. As far as we know, no spreadsheet application performs the consistency checks that are nor-

mally done for hand calculations with units. If the user puts numbers in some cells, a spreadsheet application blithely calculates results for other cells based on the formulas provided, even if the results are nonsensical from a units viewpoint. The practical and economic importance of unit consistency is great. To cite a well-known example: In 1999, the \$125 million Mars Climate Orbiter project failed because of a unit consistency error [1].

Over the past twenty years, the programming languages community has recognized the significance of units and developed techniques for dealing with them. In particular, during the 1990's, Wand [21], Kennedy [11, 13], and Goubault [9] independently developed similar methods for integrating units into ML-like languages. Unfortunately, these techniques have not yet found their way into popular programming languages. With XeLda, we have borrowed some of the ideas behind earlier research efforts, and devised some new ones, and applied them to a working implementation for a spreadsheet language.

We also tackle the problem of statically type-checking spreadsheets by analyzing the formulae in the spreadsheets following the lead of Erwig and Burnett [4]. We present a collection of rules that help identify weaknesses in spreadsheets that are likely to be errors.

Our tools for checking spreadsheets extract data from Excel, perform unit-checking and type-checking, and report errors using flow graphs drawn back into the spreadsheet. We borrow from functional programming languages and functional programming in three ways:

- we analyze a functional language, the formula language of Microsoft Excel spreadsheets;
- we adapt technology used to analyze functional languages, applying that technology to spreadsheets; and
- our tools are mostly-functional Scheme programs.

We address the important problem of unit/type consistency in the setting of spreadsheets.

Excel is weaker than spreadsheet languages such as Forms/3 [23], 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 the document is organized as follows. In Chapter 2 we describe the theoretical foundations behind the type checker. In Chapter 3, we describe how to use the unit checker tool and provide the theoretical foundations behind the unit checker. Chapter 4 describes our implementation in more detail. In Chapter 5, we report on our experience testing the checkers with some off-the-shelf scientific spreadsheets. Chapter 6 mentions related work, and Chapter 7 suggests how our tool might be improved.

Chapter 2

Types

In this chapter we describe the type checking aspect of our framework. We start with a few motivating examples that lay the groundwork for the formalism behind the type checking. We then present the judgement rules and types used to validate spreadsheets.

2.1 Motivating Examples

In this section we introduce the basic concepts and desired behavior of our type checking system by providing several examples. Suppose the spreadsheet user works at a company that produces home electronics. A very simple table from this domain is shown in Figure 2.1, where the top view shows the values in the spreadsheet, and the bottom view shows in bold the formulas and references actually entered by the user. 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 types. For example, cells B3 and B4 have both the type: TVs, so we can add them together to get another number in terms of TVs. We can also add cell B3 to C3, because they both have the type: year 2001. The result, D3, will be of type 2001, and moreover, we can abstract over the specific type of electronic device in each cell and determine that the result is also of type Electronics. On the other hand, if the user tries to add B3 and C4 (perhaps because of a formula error), this will cause a type-checking error, because these cells do not have either a year or a type of device in common.

Figure 2.2 shows a slightly more complicated table in the same domain. Here, TVs and VCRs are further subdivided into three categories each. As in the previous example, we can perform an operation (in this case, subtraction) on cells B4 and C4 because they both have type of 2001 and abstract to types of TVs. We also want to be able to handle column H, which contains the sum of defective TVs and defective VCRs. We see that cells C4 and F4 have the type 2001 in common. They are also both Electronics and both Defective, but they do not have the intermediate category of either TVs or VCRs in common. Ideally, we would like our type system to be able to capture this information by assigning the result type of 2001 and Defective Electronics.

The figure consists of two screenshots of a Microsoft Excel spreadsheet titled "Microsoft Excel - electronics.xls".

The top screenshot shows the following data:

	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

The bottom screenshot shows the same data with formulas entered in cells B3, C3, B4, C4, B5, and C5:

	A	B	C	D
1		Electronics		
2	Year	TVs	VCRs	Total
3	2001	751	522	B3+C3
4	2002	803	510	B4+C4
5	Total	B3+B4	C3+C4	B5+C5

Figure 2.1: Electronics Production by Year

Finally, consider Figure 2.3. If we follow the pattern laid out above, D3 will have types 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. In addition, we now want to be able 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 section we describe a type system that will allow us to perform all these operations, as well as preventing errors such as adding B3 to C4 in Figure 2.1 or subtracting C9 from B3 in Figure 2.3. This type system is insensitive to the specific arrangement of data, so that if the user chooses to present the data in Figure 2.3 differently (see Figure 2.4), the results of type checking will be exactly the same.

2.2 Headers and Relationships

We now describe our model of spreadsheets, defining key concepts of our type checker such as headers and relationships. We then introduce types, the basic elements of our system upon which error checking occurs. We continue with rules to govern how types may be built from spreadsheet.

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

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

Microsoft Excel - electronics.xls								
File Edit View Insert Format Tools Data Window Help								
A11 =								
A	B	C	D	E	F	G	H	
1	Electronics							
2	TVs			VCRs				
3	Year	Total	Defective	Okay	B3	C3	D3	Tot. Def.
4	2001	751	6	B4-C4	522	3	E4-F4	C4+F4
5	2002	803	7	B5-C5	510	2	E5-F5	C5+F5
6	Total	B4+B5	C4+C5	B6-C6	E4+E5	F4+F5	E6-F6	C6+F6

Figure 2.2: Electronics Production Minus Defective Products

yields a value.

A *header* is a concept that defines the common type for a group of cells. Some cells contain values that provide names for headers, and we call these *header cells*. For example, in Figure 2.1, 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 2.1, 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 2.2, 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 has two headers, TVs and 2001. In addition, there may be cells whose headers are not defined explicitly by header cells. Figure 2.3 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. The problem of inferring implicit header information will be discussed further in Chapter 5. We assume in our type-checking system that all headers are known.

There are two kinds of *relationships* that can exist between headers in our type 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

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

Microsoft Excel - electronics.xls				
File Edit View Insert Format Tools Data Window				
J1				
	A	B	C	D
1		Gross Sales		
2	Year	TVs	VCRs	Total
3	2001	149723	95176	B3+C3
4	2002	153072	93129	B4+C4
5	Total	B3+B4	C3+C4	B5+C5
6				
7		Costs		
8	A2	B2	C2	D2
9	A3	107925	77293	B9+C9
10	A4	109392	72652	B10+C10
11	A5	B9+B10	C9+C10	B11+C11
12				
13		Profits		
14	A2	B2	C2	D2
15	A3	B3-B9	C3-C9	B15+C15
16	A4	B4-B10	C4-C10	B16+C16
17	A5	B15-B16	C15-C16	B17+C17

Figure 2.3: Electronics Sales and Profits

in Figure 2.1, 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 2.3, the set of TVs *has-a* (property called) Gross Sales.

Microsoft Excel - electronics.xls				
File Edit View Insert Format Tools Data Window				
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	426245
17	Total	491100	367262	856362

Microsoft Excel - electronics.xls				
File Edit View Insert Format Tools Data Window				
I1				
	A	B	C	D
1		TVs		
2	Year	Gross	Costs	Profits
3	2001	149723	107925	B3-C3
4	2002	153072	109392	B4-C4
5	Total	B3+B4	C3+C4	D3+D4
6				
7		VCRs		
8	A2	B2	C2	D2
9	A3	95176	77293	B9-C9
10	A4	93129	72652	B10-C10
11	A5	B9+B10	C9+C10	D9+D10
12				
13		All Electronics		
14	A2	B2	C2	D2
15	A3	B3+B9	C3+C9	B15+C15
16	A4	B4+B10	C4+C10	B16+C16
17	A5	B15+B16	C15+C16	B17+C17

Figure 2.4: Electronics Sales and Profits: Alternative Layout

2.3 Types

Using the concepts of headers and relationships, we can now introduce the concept of types. Types form the basic elements upon which we perform error checking. Every cell has a type determined by the cell's headers and the relationships those headers participate in. The simplest type is the *Top* type. Any cell that has no headers has type *Top*. Examples from Figure 2.3 are cells A2 (Year) and B1 (Gross Sales). Header cells that participate in *is-a* relationships have hierarchical *is-a* types, which we denote with square brackets. The type of cell C3 in Figure 2.2 (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 types from this point onward.

Non-header cells have somewhat more complex types. The type 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 types 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 types with multiple *is-a* hierarchies using the & operator. For example, cell B3 in Figure 2.3 has two headers, 2001 and TVs. The TVs header is related to the Gross Sales header by the *has-a* relationship, so the type for B3 is:

Electronics[TVs]{Gross Sales} & Year[2001]

Note that, like other headers, the header defining the *has-a* portion of a cell's type may not be explicitly given in the spreadsheet. The tables in Figures 2.1 and 2.2, 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 type for cell B3 in Figure 2.1 is therefore similar to the previous example:

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

Now that we have covered headers and types, we focus our attention on the description of well-formed types. In particular, below, we formally state requirements for a well-formed type through judgement rules that encompass all the different kinds of cells that may appear on a spreadsheet. We observe the following conventions for notation:

- $I(d)$ is the *is-a* header for header d (possibly \emptyset)
- $U(d)$ is the type for header d
- $\mathcal{I}(a)$ is the set of *is-a* headers for the cell at location a
- $\mathcal{U}(a)$ is the type for the cell at location a
- $d \rightarrow h$ shows header d *has-a* header h
- $v(a)$ is the value of the cell at location a
- $\vec{u} (= u_1[u_2[\dots[u_n]\dots]])$ is the short-hand representation for a hierarchy of *is-a* relationships
- if $\vec{u} = u_1[\dots[u_n]\dots]$ then $\vec{u}[u'] = u_1[\dots[u_n[u']]\dots]$

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

1. Headers. The type 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 type is Top ¹:

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

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

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

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

2. Non-header cells containing values (i.e. user data), such as cell B3 in Figure 2.1. These cells also obtain types 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 type is formed by concatenating its header's type and header's name as above to obtain the *is-a* part of the type, and adding the *has-a* header at the end:

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

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:

$$\frac{\begin{array}{l} \vdash \mathcal{I}(a) = \{d, d_1, \dots, d_n\} \\ \forall i \in 1..n : \mathcal{U}(d_i) = \vec{u}_i \\ \mathcal{U}(d) = \vec{u}_d \quad d \rightarrow h \end{array}}{\vdash \mathcal{U}(a) = \vec{u}_d[d]\{h\} \& \vec{u}_1[d_1] \& \dots \& \vec{u}_n[d_n]}$$

3. Cells containing references only, such as cell E3 in Figure 2.2. The type of a cell containing a reference is the type 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 2.1. These cells contain expressions involving mathematical operators, and the resulting type for this kind of cell depends upon the actual operator in use. We discuss the rules needed for our type system to support the four basic mathematical operators ($+$, $-$, $*$, $/$) in the following section.

¹The bottom part of a judgement rule is what the type checker is able to infer based on the preconditions present in the top part of the judgement. See Pierce's book [27] for a detailed explanation of type systems.

2.4 Types and Mathematical Operators

In this section we motivate and describe the behavior of our system with regard to mathematical operations. The 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 2.1. 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 type notation this means that

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

when type-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 type. In general, we have:

$$\frac{\vdash \vec{u}_1\{h\} \quad \vdash \vec{u}_2\{h\}}{\vdash \vec{u}_1\{h\} + \vec{u}_2\{h\} \rightarrow \vec{u}_1 \oplus \vec{u}_2\{h\}}$$

Thus, when we add two types, 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 types, they must have something in common (in this case the *has-a* part). Now consider the case where the two types have a common *is-a* part. Here is a variant of the example in Figure 2.3:

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

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:

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

Or, in general:

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

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 type differ:

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

This equation clearly violates our principle stating that types must have either the *is-a* part or the *has-a* part in common in order for the addition to pass the type 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 types, and combined them by retaining in the result only the common parts of the two types. Judging from our first example, it might seem that the result of the union operation will always be a more general type than either of the two arguments. But suppose we want to perform a union operation on these two types:

$$\text{Electronics[TVs[Wide-Screen[Defective]]]} \oplus \text{Electronics[VCR[Defective]]}$$

In this case we could also say that the result should be Electronics, but we would lose information common to the two original types: 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 types creating a new type from all the common features of the two types, not just the most general ones.

To summarize, the addition rule applies only to types 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 2.2, the Okay column for TVs requires us to subtract the following two types:

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

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

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

We cannot be any more specific about the type 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 types with a common *is-a* part, as in Figure 2.3, where the data in the Profit column is given by:

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

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:

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

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, suppose we have a computation for the area of a rectangle:

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

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

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

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:

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

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 types 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:

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

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 briefly noted above, the type of cell B3 from Figure 2.1 is:

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

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

Each header conveys a distinct property for the data in the cell, which means that a well-formed & type consists of different, header inferred, types 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 types. 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,

Electronics[TVs] & Electronics[VCRs]{Gross Sales}

does not represent a valid & type 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 types. For example, in Figure 2.1, cell B5 has type:

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

which reduces to:

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

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

Electronics[TVs]{Quantity} & 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.

Chapter 3

Units

In this chapter we present the unit checker. We start with a description of the unit checker tool, XeLda. We then describe the theory underlying the unit checking mechanism.

3.1 A XeLda Tutorial

Both our unit checker and type checker use the same interface to interact with spreadsheets. Below is a description of XeLda the unit checker. The usage in the presence of types mirrors exactly that of units with the difference that cells are annotated with types (see the chapter on types for a full description) instead of units.

XeLda is a MrEd program. Figure 3.1 shows the XeLda control panel, which allows the user to specify a spreadsheet for unit-checking. Pressing the `Load File` button starts an instance of Excel on the specified spreadsheet. The user annotates cells with unit expressions, if desired, or can allow XeLda to infer unit expressions. If a cell containing just a number, and not a formula, is unannotated, it is assumed to contain a dimensionless constant. XeLda computes units for all cells with formulas, which may or may not have explicit unit annotations. Unit expressions are placed in Excel comment fields, either directly in Excel, or by using the unit annotator in the XeLda control panel. The annotator assigns a unit expression for all cells within a “cell range,” or a block of cells. A cell range can be entered textually in the control panel, or selected with the mouse in Excel itself.

Once the spreadsheet is loaded and annotated, the user presses the `Analyze` button to perform unit-checking. Two basic kinds of unit errors can occur:

- a unit expression derived from a formula appearing in a cell is different from its unit expression annotation (a *match error*), or
- the unit expression derived from a formula indicates an error in the formula or one of its subexpressions (a *consistency error*).

XeLda flags these errors by coloring the cells where they occur, orange for a match

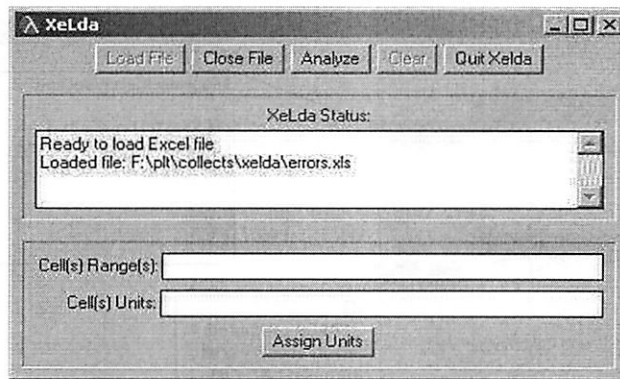


Figure 3.1: XeLda control panel.

error and yellow for a consistency error. In addition, whenever an error occurs at a cell, all cells that depend on it are colored purple, indicating *error propagation*.

When a unit error occurs, it is useful to know *why* the error occurred, and what the *sources* of the error are. XeLda gives an explanation of why an error has occurred in a cell by inserting text in the cell's comment field beneath its unit expression, if any. Unit expressions are entered using the syntax shown. Error explanations are prefaced by a semicolon. The sources of an error are shown by drawing arrows to the error cell from the cells it depends on.¹ Right-clicking on an error cell draws its source arrows and colors the source cells red; right-clicking again removes the arrows and coloring. Figure 3.2 shows a XeLda-analyzed spreadsheet with a textual explanation display and some source arrows drawn. Cell C4, annotated with *kilogram-meter/second²* is the product of cells A2 and B3, annotated with *kilogram* and *meter-second²*, respectively. Therefore, there is a mismatch between the computed units for C4 and its units annotation. The other error occurs at cell C9, which is unannotated, because cells A7 and B8 are annotated with *apple* and *orange*, respectively. In both cases, the data sources for the cells where the error is detected are indicated by arrows from the sources to those cells.

Some cells may depend on a range of cells for their computation. For example, if an error cell is an element of a matrix resulting from a matrix multiplication, Excel shows dependency arrows from both source matrices.

The **Clear** button on the XeLda control panel allows the user to remove error messages, error coloring, and source arrows. Removing those artifacts from a spreadsheet supports iterative debugging of unit errors. If XeLda detects many errors in a spreadsheet, the user might wish to fix just a few of them at a time over multiple sessions. The user can clear a worksheet from XeLda, save it without any extraneous information added by the analysis, and work on it at a later time.

We switch our attention now to the unit checker. As mentioned before, a spreadsheet has cells that may contain formulas and unit expression annotations. We begin

¹The idea of drawing source arrows came from the MrSpidey [7] static debugger for PLT DrScheme.

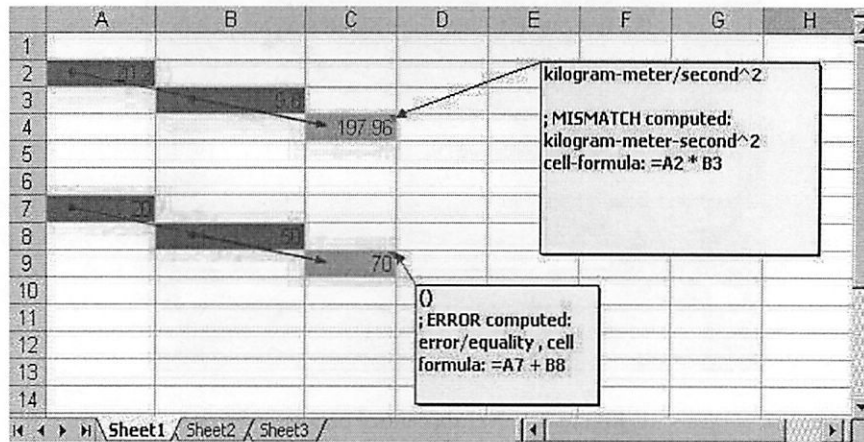


Figure 3.2: Unit errors in a spreadsheet.

with a detailed description of formulas.

The abstract syntax of *formulas* is given by:

$$e ::= n \mid \text{cell-ref} \mid \text{id} \mid e \text{ op } e \mid \text{fun}(e, \dots, e)$$

where n is a number, *cell-ref* is a cell reference, *op* is an arithmetic operator, and *fun* is an identifier denoting an Excel function. Examples of Excel functions are SUM, AVERAGE, and MAX. Lexically, a cell reference is a string consisting of one or two letters followed by one or more digits, where the digits denote a positive number; examples are A1, B52, and CH22. Excel imposes some restrictions on the letters that may be used in cell references, but those need not concern us here. An *id* is an identifier that names a spreadsheet cell, but is lexically not a cell reference. It is possible for a function name to do double-duty as the name of a cell; the role of an *id* can be determined by syntactic context. The association between cell references and particular spreadsheet cells is implicit in the layout of the spreadsheet. Cell names, on the other hand, are assigned by users. Excel will flag an error if an identifier used in a formula is not associated with any cell, so XeLda need not consider that possibility. In the concrete syntax of formulas, parentheses may be used for grouping, of course, and there are alternate notations for cell references. With this grammar, we purposely ignore a few constructs found in Excel formulas, such as boolean constants and conditionals.

Unit expressions represent the dimensions associated with a numeric value, consisting of a possibly empty list of unit, exponent pairs, or an error:

$$U ::= ((w \ n) \dots) \mid \text{error/equality} \mid \text{error/propagate} \mid \text{error/circular}$$

where each w is the name of a unit and n is an integer exponent. The unit expressions of the form *error/...* are *error unit expressions*. Our implementation uses a more conventional notation for units, rather than the Scheme-like list syntax given

here. The names of units are arbitrary; examples are kilogram or second. In our system, we do not associate those names with physical dimensions; their significance is simply the user's interpretation of them.² The empty list denotes dimensionlessness. A nonempty list of units and their exponents denotes a product of units; a negative exponent indicates division. For example, the unit expression

`((kilogram 1) (meter 1) (second -2))`

denotes an SI Newton.

Because we wish to compare unit expressions, it is convenient to have a normal form for them. Within a unit expression, the ordering of units is unimportant: a kilogram-meter denotes the same unit as a meter-kilogram. A unit name needs to appear within a unit expression only once, because the exponents in multiple occurrences can be summed. Any unit with a zero exponent does not contribute to the unit expression, and therefore can be omitted. So:

Definition 1 *A unit expression $((w_1 n_1) \dots (w_m n_m))$ is in normal form iff*

- *each w in w_1, \dots, w_m is distinct;*
- *$w_i \leq w_{i+1}$ for $1 \leq i < m$, where the comparison on the w 's is lexicographic; and*
- *$n_j \neq 0$ for $1 \leq j \leq m$.*

This definition is close to Kennedy's presentation of unit expressions as elements of an Abelian group [13], except for the sorting requirement, which is useful for an implementation. Clearly, we can obtain the normal form of any unit expression by summing exponents of like units, filtering out units with a zero exponent, and sorting on the unit names.

3.2 Deriving Units

Given a spreadsheet, we wish to compute unit expressions for each cell containing a number or formula that yields a number. Our basic strategy is to take the unit expressions of arguments to a function and combine them according to that function. That strategy corresponds to the way people derive units for paper calculations.

We can partition nonempty spreadsheet cells into those that contain a number (*value cells*) and those whose value is derived from a formula (*formula cells*). In Excel, an equals sign as the first character in a cell introduces a formula cell; all other nonempty cells contain values. A formula can be extremely simple, such as `=17`. We consider formulas that consist of just an equals sign followed by a number as a value cell. Cells whose text is of the form `=name` or `=A5` are formula cells, because their value depends on other cells, even though they do not contain operators or functions.

The distinction between functions and operators is syntactic; from here on, we refer to both as just "functions". Some Excel functions, such as SUM and AVERAGE,

²While we might get stronger unit-checking by fixing the available unit names, XeLda allows users to choose any consistent system of units. New units can come into existence; consider the euro.

are variadic. To indicate an Excel function, we subscript its name, writing the arithmetic operators as $+_{XL}$, $*_{XL}$, and so on, and the others just mentioned as SUM_{XL} , and $AVERAGE_{XL}$.

Except in the case of circular references, described below, we use *unit transformers* to compute unit expressions. The use of a unit transformer mimics the derivation of units that people perform when doing hand calculations. For each spreadsheet function or operation Fun_{XL} , we introduce the unit transformer \widehat{Fun} , which takes one or more unit expressions and produces a unit expression. The arity of \widehat{Fun} is the same as Fun_{XL} ; \widehat{Fun} is variadic if Fun_{XL} is.

For example, $+_{XL}$ performs addition in Excel. Let U be any unit expression. $error-unit(U)$ holds iff U is an error unit expression. We define the unit transformer:

$$U_1 \hat{+} U_2 = \begin{cases} \text{error/propagate} & \text{if } error-unit(U_1) \text{ or } error-unit(U_2) \\ U_1 & \text{if } U_1 = U_2 \\ \text{error/equality} & \text{otherwise} \end{cases}$$

That is, the units associated with the arguments to $+$ must be identical and not error units; otherwise, we have a units-equality error. The $\hat{+}$ just mimics what people do when adding two numbers with associated units: they check that each addend has the same units, and if so, assign that unit to the sum; otherwise, there is an error. The $\hat{-}$ unit transformer for subtraction is identical to $\hat{+}$.

Now consider multiplication as performed by the $*_{XL}$ operator. The unit transformer for multiplication is:

$$U_1 \hat{*} U_2 = \begin{cases} \text{error/propagate} & \text{if } error-unit(U_1) \text{ or } error-unit(U_2) \\ [U_1 @ U_2] & \text{otherwise} \end{cases}$$

where $@$ indicates list append, and $[\cdot]$ indicates normalization.

The unit transformer for division has a slight twist:

$$U_1 \widehat{\text{div}} U_2 = \begin{cases} \text{error/propagate} & \text{if } error-unit(U_1) \text{ or } error-unit(U_2) \\ [U_1 @ \overline{U_2}] & \text{otherwise} \end{cases}$$

where $\overline{U_2}$ is like U_2 , with all the signs of exponents reversed. In Excel, division is denoted by $'/'$, as usual; here, we write div to avoid syntactic confusion with other slashes.

In the ordinary case, the unit expression for a formula is derived bottom-up, starting at value cells, and propagating up to formula cells. Therefore, we might characterize

unit expression derivation as the computation of synthetic attributes of parse trees. In order to lessen the annotation burden for users, if a leaf cell does not have an explicit annotation, we assume it is annotated with the empty list. For cells with formulas, we use a unit transformer to provide a derived unit expression. Define a map from formula expressions to unit expressions:

$$Units(e) = \begin{cases} () & \text{if } e \text{ is a number} \\ Anno(e) & \text{if } e \text{ is a value cell reference} \\ Anno(e') & \text{if } e \text{ is an identifier naming the value cell } e' \\ Units(Formula(e)) & \text{if } e \text{ is a formula cell reference} \\ Units(Formula(e')) & \text{if } e \text{ is an identifier naming the formula cell } e' \\ \widehat{Fun}(Units(arg_1), \dots, Units(arg_n)) & \text{if } e \text{ is } Fun_{XL}(arg_1, \dots, arg_n) \end{cases}$$

where

- *Anno* is a map from cell references to unit expressions, indicating user-supplied unit annotations, and
- *Formula*(*e*) is the formula associated with the cell reference *e*

This definition is well-founded for finite formulas. Some Excel formulas depend on their results, allowing iterative computations; we consider circular references below.

Unit transformers are only useful if they compute a sensible result. The transformers we have just seen, $\widehat{+}$, $\widehat{-}$, $\widehat{*}$, and \widehat{div} given are fundamental, because they correspond to basic arithmetic operations, and they handle units in the expected way for those operations. Let Fun_{XL} be a function that takes some numbers and produces a number by uses of $+_{XL}$, $-_{XL}$, $*_{XL}$, and div_{XL} . Then we derive \widehat{Fun} , a unit transformer for Fun_{XL} , by substituting the corresponding unit transformer for each arithmetic operation. We can define this notion in a lambda calculus extended with numbers, arithmetic operations, and corresponding unit transformers.

Definition 2 (Unit transformer) Let f be the term $\lambda x.e$, where e may contain arithmetic operations, but not $\widehat{+}$, $\widehat{-}$, $\widehat{*}$, or \widehat{div} . Then

$$\widehat{f} \equiv \lambda x.e[\widehat{+}/+, \widehat{-}/-, \widehat{*}/*, \widehat{div}/div]$$

is a unit transformer for f .

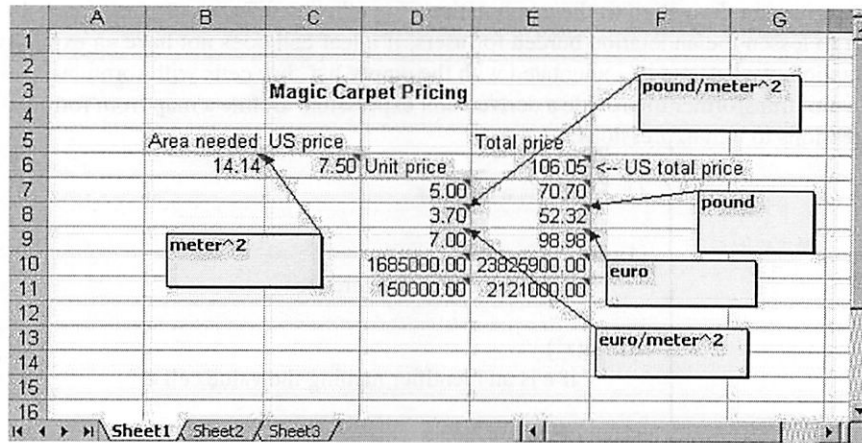


Figure 3.3: Element-polymorphic table map.

Concretely, suppose AVERAGE_{XL} when applied to three arguments is implemented as

$$\lambda x_1 x_2 x_3. (x_1 + x_2 + x_3) \text{ div } 3$$

then the corresponding unit transformer $\widehat{\text{AVERAGE}}$ would be

$$\lambda x_1 x_2 x_3. (x_1 \hat{+} x_2 \hat{+} x_3) \widehat{\text{div}} 3$$

Of course, for Excel functions, no program text is available. Because we do not have program text, we intend this specification to be notional, rather than literal, for our implementation.

3.3 Tables and Matrices

Excel supports tables, typically used to perform “what-if” analyses, and matrix operations, both of which are handled specially by XeLda. We cannot use unit transformers to compute unit expressions in quite the same way as in the ordinary case, because Excel does not associate formulas with the result cells.

Tables

A *data table* in Excel is a range of cells that shows the results of a formula computation based on different input values. Tables are derived from ranges of cells much as functional programmers derive lists by mapping functions over input lists. Using tables, one can calculate multiple variations of an operation and view them as a block of cells.

Figure 3.4 shows a simple data table. Cell A1 contains the number 10, and cell C1 contains the formula $A1 + 1$, so the number 11 is displayed. The range B2:B10 contains some other numbers. Each cell in the range C2:C10 contains the formula $\{=\text{TABLE}(, A1)\}$. Excel “maps” the $x + 1$ operation for each value in B2:B10

and places it in the neighboring cell in C2:C10. When performing table operations, Excel uses the layout of the spreadsheet to determine what input values to use, and where to put the results.

	A	B	C	D
1	10		11	
2		17	18	
3		18	19	
4		23	24	
5		27	28	
6		31	32	
7		43	44	
8		47	48	
9		49	50	
10		58	59	
11				

Figure 3.4: Data table.

Tables accept formulas with either unary or binary arity, indicating how many numbers are replaced in the calculation. In the preceding example, just A1 is replaced, so the formula is of unary arity. Currently, XeLda analyzes only unary formulas, but handling binary arity ones should not be difficult. Depending on the placement of the comma in the TABLE arguments, table results may be placed in an row adjoining the input values, as in the example just given, or in an adjoining column.

For each cell in a table, we compute a unit expression by applying the unit transformer for the cell containing the formula, except that we use the unit expression for the current input value instead of that for the original input. Using the table in Figure 3.4, suppose we wish to calculate a unit expression for the cell C3 in the table. The corresponding input cell is B3. The unit expression for C3 is the result of applying the unit transformer $\hat{+}$ to the unit expression for B3 and the empty list, the unit expression for the constant 1.

Because a unit transformer is applied to each input element, which may have different unit annotations, elements of a result table may have different associated unit expressions. For each input element, we always use the same unit transformer; hence the computation of units for tables yields true parametric polymorphism over unit expressions. To continue the analogy with mapping over lists, our approach to tables is comparable to mapping a function over a heterogeneous list and obtaining heterogeneous results. For an example of such polymorphism, see Figure 3.3. We wish to compute the cost of a magic carpet in several world currencies. The area of the carpet is specified in square meters. The original input to the table is the price of a magic carpet in dollars per square meter. The “what if” inputs to the table are prices per square

	A	B	C	D	E
1	Physics 101: Linear Motion (linearly-increasing acceleration)				
2	Distance:	Speed:	Acceleration:		
3	$x = x + v * dt + a * dt^2 / 2$	$v = v + a * dt$	$a = a + da * dt$		m/s^2
4	0.1015	0.21	0.3		
5					
6	m	da = 0.2			m/s^3
7		dt = 0.1			
8					
9		m/s			
10			s		
11					
12					

Figure 3.5: Equation of motion.

meter, where the price varies by country. In the resulting table, each numeric price is annotated with a unit expression indicating the appropriate currency.

Matrix values

Excel supports operations that result in matrix values, for example, matrix multiplication. In Excel, a matrix value occupies a rectangular block of cells. Each cell in the matrix contains the formula that produced the result. In the case of matrix multiplication, that formula has the form $\{=MMULT(M1, M2)\}$ where M1 and M2 are cell ranges denoting matrix arguments.

For each cell in the result matrix, we check that all elements in the row in M1 required for the calculation of that cell have equal unit expressions. Similarly, we check the units for the column in M2 that produced the entry for equality. If any of those input elements has an associated error unit, the unit expression for the result cell is *error/propagate*. If either of the equality checks fails, the result is *error/equality*. Otherwise, the unit for the cell becomes the normalized product of the unit expressions for the row in M1 and the column in M2.

We can formalize this idea with a unit transformer for matrix multiplication that is *specialized* to a result cell at row i and column j :

$$\widehat{MMULT}(M_1, M_2)(i, j) = \begin{cases} \text{error/propagate} & \text{if ErrorRowUnits}(M_1, i) \text{ or } \text{ErrorColUnits}(M_2, j) \\ U_{1,i} @ U_{2,j} & \text{if UniformRowUnits}(M_1, i) \text{ and } \text{UniformColUnits}(M_2, j) \\ \text{error/equality otherwise} & \text{otherwise} \end{cases}$$

where

- $U_{1,i}$ is the unit expression associated with each cell in row i in M_1 ,
- $U_{2,j}$ is the unit expression associated with each cell in column j in M_2 ,
- `ErrorRowUnits` and `ErrorColUnits` hold iff any unit expression associated with the given row or column is an error unit expression,
- `UniformRowUnits` and `UniformColUnits` hold iff all unit expressions associated with the given matrix row or column are identical.

3.4 Circular References

Excel allows formulas to depend on themselves, though it issues a warning to the user when they are entered. The simplest case is a cell reference that depends on itself, so that a cell, say A1, contains the formula A1. When a formula is circular, Excel computes a solution iteratively; unless otherwise specified, cells start off with a value of 0. A user-selectable limit on iteration enforces termination, even in the absence of a fixpoint; the user can request successive rounds of iteration. XeLda is able to infer unit expressions and detect unit errors in the presence of circular references with a specialized unit inference mechanism.

Figure 3.5 shows a spreadsheet containing three circular references. We wish to iteratively compute the position of a particle under a linearly-increasing acceleration, given an initial position, velocity, and acceleration. Cells B6 and B7 specify the increments for acceleration and time, respectively. With each round of iteration, we compute a new position (cell A4), velocity (B4), and acceleration (C4). By varying the iteration limit in Excel, we can vary the time interval used to compute the new values. XeLda is able to validate the units we have assigned to the cells in this spreadsheet.

During parsing, XeLda is able to distinguish formulas with circular dependencies from those with purely tree dependencies. We first solve for unit expressions for the latter class of formulas, using unit transformers as described above. Next, we derive unit expressions for the formulas with circular dependencies.

For formulas with circular references, unit derivation is a three-step process. First, for each circular formula, we generate a set of *constraints* containing *unit variables* and unit expressions. Next, we build equivalence classes of unit variables and propagate class representatives to other constraints. We are left with constraints that we map into algebraic equations, which are then transformed into a set of homogeneous linear equations. Finally, we solve for the unit variables using Gaussian eliminations, yielding unit expression solutions for the circular formulas.

3.4.1 Constraint Generation

In the case of formulas with circular dependencies, we cannot apply unit transformers, because dependency loops would lead to divergence. Instead, for each application of an Excel function, we generate constraints on unit expressions appropriate to that function. Essentially, constraint generation postpones application of unit transformers.

From the definitions of $\widehat{+}$, $\widehat{-}$, $\widehat{*}$, and $\widehat{\text{div}}$, the unit transformers for arithmetic operations, and Definition 2, we see that, except in error cases, all unit transformers produce unit expressions that contain only the units appearing in their inputs. Both $\widehat{+}$ and $\widehat{-}$ impose *equality constraints* on their inputs; $\widehat{*}$ and $\widehat{\text{div}}$ specify how to combine input units to obtain a result unit expression; such specification yields *append constraints*. Other unit transformers may specify both equality and append constraints.

Within each formula containing a circular reference, for each use of an Excel function, we provide fresh unit expression variables for the application node and its arguments and generate constraints. For the arithmetic operators, we generate constraints as follows:

$$\begin{aligned} e_1 +_{XL} e_2 &\Rightarrow \begin{cases} \alpha = \alpha_{e_1} \\ \alpha = \alpha_{e_2} \end{cases} \\ e_1 -_{XL} e_2 &\Rightarrow \begin{cases} \alpha = \alpha_{e_1} \\ \alpha = \alpha_{e_2} \end{cases} \\ e_1 *_{XL} e_2 &\Rightarrow \{ \alpha = [\alpha_{e_1} @ \alpha_{e_2}] \} \\ e_1 \text{div}_{XL} e_2 &\Rightarrow \{ \alpha = [\alpha_{e_1} @ \overline{\alpha_{e_2}}] \} \end{aligned}$$

where α is the unit expression variable associated with the formula. We also generate equality constraints for references to annotated value cells, and to cells with noncircular formulas, whose units have already been computed. Hence, equality constraints can have nonvariable unit expressions on their right-hand sides.

Let us look at the constraints generated from an arithmetic formula in our example in Figure 3.5. Cell C4 contains the circular formula: $C4 + A6 * A7$. The constraints generated for this formula are:

$$\begin{aligned} \alpha_{C4} &= \alpha_1 \\ \alpha_{C4} &= \alpha_2 \end{aligned}$$

where α_1 and α_2 are fresh variables for the operands of $+$. For the subformulas, we obtain: with the formula in cell C4 are:

$$\begin{aligned} \alpha_1 &= \alpha_{C4} \\ \alpha_2 &= [\alpha_3 @ \alpha_4] \\ \alpha_3 &= ((m\ 1)\ (s\ -3)) \\ \alpha_4 &= ((s\ 1)) \end{aligned}$$

where α_3 and α_4 are associated with the arguments of $*$.

For each application of other Excel functions, we generate constraints in a similar fashion. Notionally, we generate constraints guided by the syntax of the function's definition. As for unit transformers, we can use the lambda calculus for illustration. For the application $(\lambda x.e) e'$, we generate constraints for the uses of the arithmetic operators in e according to the specification above, and the following additional equality constraints:

$$\begin{aligned} \alpha &= \alpha_e \\ \alpha_{e'} &= \alpha_x \end{aligned}$$

```

(provide
  load-xl-file          ;; filename -> void
  close-xl-workbook     ;; void -> void
  clear-cell-precedents ;; cellref -> void
  get-cell-text         ;; cellref -> string
  get-cell-value        ;; cellref -> cell-value
  get-cell-formula      ;; cellref -> string
  get-cell-name         ;; cellref -> string
  get-cell-row-col      ;; cellref -> (list num num)
  get-cell-color        ;; cellref -> color
  set-cell-color!       ;; cellref color -> void
  get-cell-comment      ;; cellref -> string
  set-cell-comment!     ;; cellref string -> void
  delete-cell-comment!  ;; cellref -> void
  iterate-over-worksheet
    ;; (cellref -> V) (V -> bool) ->
    ;; (listof (list cellref V))
)

```

Figure 3.6: I/O layer interface.

where $\alpha_{e'}$ is associated with the occurrence of e' , and α_x is associated with all instances of the binding x . As for unit transformers, this specification is notional, because we do not have access to the code used to implement Excel functions.

3.4.2 Constraint Resolution

In order to solve the constraints, we use the following algorithm. Because each transformation substitutes equals for equals, they are sound.

From the equality constraints, we generate equivalence classes of variables. Choose a representative variable for each class. For each class, if at least two members participate in constraints with distinct nonvariable right-hand sides, then we choose as the *representative unit* for the class *error/equality*. If there is exactly one nonvariable unit expression associated with the class, choose that unit expression as the representative unit. Otherwise, there is no representative unit for the class. Substitute the representative unit, if any, for occurrences of class members on the right-hand sides of append constraints; otherwise substitute the representative variable. Substitute the representative variable for occurrences of class members on the left-hand sides of the append constraints.

Next, we deal with the append constraints. Each of these constraints has the form:

$$\alpha_i = [\beta_1 @ \beta_2] \quad (3.1)$$

where β is a metavariable ranging over unit variables and unit expressions.

A unit expression U may be written in algebraic form. Without loss of generality, suppose U is in normal form. Then its algebraic form is given by:

$$u = \prod_i w_i^{n_i} \quad (3.2)$$

where each w_i is a distinct unit name; in this algebraic setting, consider these to be constants.

From equation 3.1, we can have unit variables, unit expressions, or both, on the right-hand sides of append constraints. We can represent every append constraint in the following algebraic forms:

$$\alpha = \alpha_1 \times \alpha_2 \quad (3.3)$$

$$\alpha = \alpha_1 \times u_1 \quad (3.4)$$

$$\alpha = u_1 \times u_2 \quad (3.5)$$

Because normalization is a presentation issue, rather than semantic, it does not appear in the algebraic representation. The \times operator is commutative, so the order of its arguments does not matter. We wish to solve for the unit variables in terms of unit expressions. In one special case, we simplify before proceeding to Gaussian elimination. If the left-hand side unit variable also occurs on the right-hand side of equations of type 3.4, we can immediately deal with the constraint by examining the right-hand side unit expression u . If u is the empty list, we discard the constraint, because there is no effective constraint on the variable. Otherwise, associate `error/circular` with the variable, because there is no solution for the constraint.

For the remaining constraints, we divide through by their left-hand sides. The equations 3.3 – 3.5 become:

$$1 = \alpha^{-1} \times \alpha_1 \times \alpha_2 \quad (3.6)$$

$$1 = \alpha^{-1} \times \alpha_1 \times u_1 \quad (3.7)$$

$$1 = \alpha^{-1} \times u_1 \times u_2 \quad (3.8)$$

Taking logarithms, we get:

$$0 = -\log \alpha + \log \alpha_1 + \log \alpha_2 \quad (3.9)$$

$$0 = -\log \alpha + \log \alpha_1 + \log u_1 \quad (3.10)$$

$$0 = -\log \alpha + \log u_1 + \log u_2 \quad (3.11)$$

From equation 3.2, by taking the logarithm of a unit expression in algebraic form u , we have:

$$\log u = \sum_i (n_i \times \log w_i)$$

Substituting for the logarithms of unit expressions in equations 3.9 – 3.11, we obtain the following *linear equations*:

$$0 = -\log \alpha + \log \alpha_1 + \log \alpha_2 \quad (3.12)$$

$$0 = -\log \alpha + \log \alpha_1 + \sum_i (n_{1_i} \times \log w_{1_i}) \quad (3.13)$$

$$0 = -\log \alpha + \sum_i (n_{1_i} \times \log w_{1_i}) + \sum_j (n_{2_j} \times \log w_{2_j}) \quad (3.14)$$

We solve these equations using Gaussian elimination. If we have fewer equations than variables, we attempt to solve for as many variables as possible. The remaining variables are unconstrained, so we assign them `error/circular`.

For each unit variable α that may have a solution, Gaussian elimination produces equations of the form:

$$c \log \alpha = \sum_i (n_i \times \log w_i)$$

where c is a nonzero integer. Equivalently:

$$\alpha = \prod_i w_i^{n_i/c}$$

We accept only solutions where all n_i/c are integers. In all other cases, we assign α the unit expression `error/circular`.

Let us see how this approach applies to our Figure 3.5. After constraint generation and equivalence class substitution we have the following append constraints to solve:

$$\begin{aligned} \alpha_{A4} &= [\alpha_{B4} @ ((s \ 1))] \\ \alpha_{A4} &= [((m \ 1) (s - 2)) @ ((s \ 2))] \\ \alpha_{B4} &= [((m \ 1) (s - 2)) @ ((s \ 1))] \end{aligned}$$

Converting these to algebraic equations we obtain:

$$\begin{aligned} \alpha_{A4} &= \alpha_{B4} \times s \\ \alpha_{A4} &= m \\ \alpha_{B4} &= m \times s^{-1} \end{aligned}$$

We have one equation of type 3.4 and two of type 3.5. Following the steps outlined above, we get the system of linear equations

$$\begin{aligned} 0 &= -\log \alpha_{A4} + \log \alpha_{B4} + \log s \\ 0 &= -\log \alpha_{A4} + \log m \\ 0 &= -\log \alpha_{B4} + \log m - \log s \end{aligned}$$

By Gaussian elimination we have:

$$\begin{aligned} \log \alpha_{A4} &= \log m \\ \log \alpha_{B4} &= \log m - \log s \end{aligned}$$

hence

$$\begin{aligned} \alpha_{A4} &= m \\ \alpha_{B4} &= m/s \end{aligned}$$

```

;; (cellref -> V) (V -> bool) -> (listof (list cellref V))
;; iterate f over cells in a worksheet, returning a list of
;; cell value pairs, where those values satisfy pred?
(define (iterate-over-worksheet f pred?)
  (let* ([used-range (com-get-property worksheet "UsedRange")]
        [cells (com-get-property used-range "Cells")]
        [first-row (com-get-property cells "Row")]
        [first-col (com-get-property cells "Column")]
        [num-rows (com-get-property cells "Rows" "Count")]
        [num-cols (com-get-property cells "Columns" "Count")]
        (filter (lambda (entry) (pred? (cadr entry)))
              (apply append
                    (build-list num-rows
                              (lambda (row)
                                (build-list num-cols
                                          (lambda (col)
                                            (let ([curr-row (+ row first-row)]
                                                  [curr-col (+ col first-col)])
                                              (list (cell-name curr-row curr-col)
                                                    (f (get-cell curr-row curr-col)))))))))))

```

Figure 3.7: MysterX: higher-order COM programming.

as desired.

We could use this technique for handling circular references even in the ordinary case. We do not do so for several reasons:

- the use of unit transformers corresponds more closely to manual manipulation of units;
- using constraints is more complex and computationally expensive;
- perhaps most importantly, the source of errors becomes unclear.

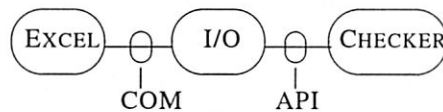
When deriving units via a unit transformer, the units for a given expression are derived from the units of its subexpressions. When solving constraints, we use equational reasoning, which masks the sources of derived unit expressions.

Chapter 4

Implementation

XeLda is a program that runs in the DrScheme programming environment [6, 8]. The control panel relies on DrScheme's MrEd graphical classes. We use the MysterX COM extension for DrScheme to interact with Excel [20].

The following diagram captures the high-level XeLda system architecture:



Currently our interface works with either the unit checker or the type checker, but not with both simultaneously. The main reason is the current requirement that spreadsheets must be annotated with either units or types in order to be checked.

An I/O layer mediates between the checker(units or types) and Excel, handling all COM operations. That way, the checker does not have to deal with low-level details. The interface exported by the I/O layer, shown in Figure 3.6, suggests the operations used by the checker.

4.1 Excel, COM, and MysterX

Excel exposes all of its functionality through a variety of Component Object Model (COM) interfaces. To access those interfaces, typically, Excel users write code in Visual Basic for Applications (VBA), executing the code as a spreadsheet "macro." Any COM client in any programming language may access Excel through those interfaces, though. XeLda uses the MysterX extension to PLT Scheme to communicate with Excel.

The Excel COM programming model is complex: in Excel 2002, there are over 400 COM interfaces available, each with several methods and properties. In XeLda, we use only a few of them, such as `_Application`, `_Workbook`, `_Worksheet`, and `IRange`. By invoking their methods and using their properties, COM can be used to make Excel do anything a person might do by interacting with the application,

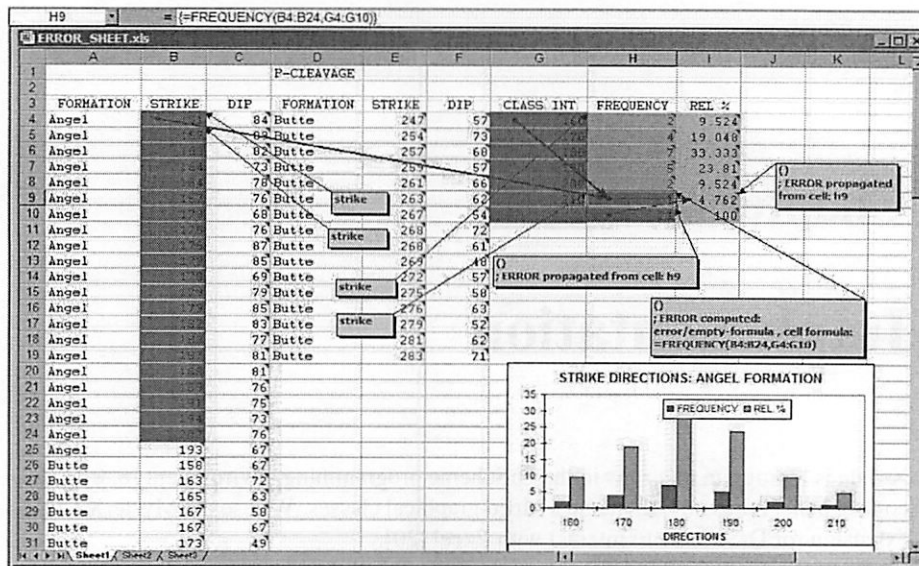


Figure 4.1: Unit errors in off-the-shelf spreadsheet.

such as: entering and retrieving information in cells; opening files; drawing charts; following hyperlinks; invoking VBA macros; and so on. Some of Excel's interfaces are *outbound*, allowing Excel to call code in a client event sink. For example, the outbound `WorkbookEvents` interface has a `BeforeClose` event that is invoked when a workbook closes.

COM interfaces are not directly visible to the `MysterX` programmer. Instead, the programmer invokes methods and gets and sets properties of Scheme objects that represent COM objects. To retrieve a cell range from a worksheet, for example, we could use

```
(define range
  (com-get-property worksheet "Range" "D4"))
```

assuming that `worksheet` is already bound to a worksheet object. In this case, the range is a single cell, D4. To add a comment to that range, we could use

```
(com-invoke range "AddComment"
  "this is a comment")
```

4.2 Mostly-Functional COM Scripting

Using Scheme makes COM programming pleasant. For instance, the I/O module exports a function `iterate-over-worksheet`, which gathers information about the current worksheet. Its inputs are a function that extracts information from a cell, and

a predicate on that information, indicating whether the cell is of interest. To give the flavor of programming with MysterX, we show the definition of the function in Figure 3.7.

The function body is a doubly-nested loop that runs over the rows and columns of the used portion of a worksheet, building a result list and filtering out undesired elements. Note that the MysterX primitive `com-get-property` can take several arguments to form a *path* of properties, as in the bindings for `num-rows` and `num-cols`. That is, the `Rows` and `Columns` properties of cell ranges return COM objects; we obtain the `Count` property of those objects within a single call to `com-get-property`. The function returns a list of symbol, value pairs, where the symbol is a cell name and the values are those extracted by the passed-in function. Using this function in the SOLVER module, we can easily obtain a list of worksheet cells and their formulas with the function application:

```
(iterate-over-worksheet
  get-cell-formula
  (lambda (formula)
    (and (> (string-length formula) 1)
         (char=? (string-ref formula 0)
                  #\=))))
```

The first functional argument retrieves the formula from a cell. The second argument is the predicate that indicates whether we really have a formula, by making sure the formula text is not the empty string and that it begins with an equals sign.

Chapter 5

Applying the Checkers

We have tested the checkers on our own spreadsheets for testing purposes, and also on some scientific spreadsheets occurring “in the wild.” For the latter class of spreadsheets, we wished to see if we could detect any errors, or at least confirm unit/type consistency.

The book by Filby [5] has an accompanying CD-ROM that provides spreadsheets from a variety of scientific disciplines. None of the spreadsheets appearing in the book has units or types directly associated with spreadsheet cells. Many of the spreadsheets, though, specify units in textual headers for columns and rows containing numeric data. Using XeLda’s unit annotation feature, we inserted unit expressions in the cells labelled by those headers.

At the same time, we annotated the value cells in the spreadsheets appearing in the book with the types that we should have obtained through header inference. Header inference is a difficult artificial intelligence and natural language processing problem. Our current implementation does not have a type inference mechanism. Instead, it relies on the same process used by the unit checker, where the user has to annotate the value cells with the correct types. In the future we plan to work on automating the type inference, and as part of that mechanism, we may do semantic analysis of headers to determine relationships with the aid of WordNet [29].

The table in Figure 5.1 describes the spreadsheets we used to test our checkers. Each horizontal grouping represents one Excel file; within each grouping, each line represents a worksheet. The size given is the number of non-empty cells; the times are in minutes and seconds, as provided by PLT Scheme’s `time` macro. The *COM* time represents the total time took by the application minus the time the checker (units/types) took. The *Units* represents the time for the unit checker and the *Types* represents the time for the type checker. The mark (✓) accompanying the checkers times represents those spreadsheets for which the checkers found errors.

The discrepancy between the checkers times and the *COM* time is explained by the slowness intrinsic in the COM Automation method which we have used to communicate with Excel. There is only one spreadsheet, the Volterra-Lotka Model, that took a considerable amount of time to unit check. We speculate that this is due to the formula structures present in this spreadsheet. Most of the formulas are additions and

Author	Description	Size	COM	Units	Types
S. Leharne	Acid Base Titration	109	0:23	0:01	0:01
W.J. Orvis	Oscillations Frequency	43	0:18	0:08 ✓	0:01
	Oscillations Euler Method	345	1:51	0:27 ✓	0:01
A.A. Gorni	Cubic Crystalline Systems X-Ray Diffraction	83	0:39	0:04	0:01
W.J. Orvis	Electron Drift Velocity in GaAs	44	0:15	0:03	0:01
J.P. LeRoux	Cleavage Strike Direction	236	1:13	0:06 ✓	0:03 ✓
	Palaeocurrent	284	1:38	0:04	0:02
	Untilt	53	0:21	0:03	0:01
	Chi-square	41	0:15	0:01	0:01
A.A. Gorni	Grain size of microstructure	40	0:22	0:02	0:01
E. Neuwirth	Feigenbaum Diagram	1000	2:57	0:02	0:01
E. Neuwirth	Simple Model	54	0:06	0:01	0:01
	Parametric Model	55	0:09	0:02	0:01
	Complex Model	56	0:12	0:01	0:01
	Complex Model with Table	75	0:18	0:02	0:01
	Complex Model with Stepwidth	57	0:07	0:02	0:01
	Volterra-Lotka Model	8004	14:38	3:00	0:21
	Planets	4001	12:18	0:12	0:16
	Planets Halfstep	4001	10:10	0:10	0:14
W.J. Orvis	Blackbody spectral emission	507	0:52	0:01	0:01
A.A. Gorni	Viscometric molecular weight	41	0:46	0:03	0:01
A.A. Gorni	Point count method	26	0:17	0:02	0:01

Figure 5.1: Experimental results.

subtractions. To unit check those we have to check that each operand unit is identical across both addition and subtraction. None of the other spreadsheets had addition and subtraction formulas on the same order of magnitude as this one and this might be the reason why the rest took far less time to unit check.

Three of the tested worksheets had unit errors detected by the unit checker. For the Oscillation worksheets, the errors were caused by inappropriate textual labelling of units by the author. By using those same units in annotations, XeLda detected unit inconsistencies. The other error found was caused by supplying too big a cell range to the Excel FREQUENCY function. Figure 4.1 shows that spreadsheet. The FREQUENCY function is used in the formula for each of the shaded cells in column I. That function takes two vectors of numbers, where the second is in increasing order, indicating bins in which to place numbers from the first vector. It returns a vector that contains the number of numbers from the first vector within each bin; the last element in the returned vector is the number of numbers greater than the highest bin. In the spreadsheet shown, the cell range for the second argument erroneously includes the cell G10, which has been left blank and has no unit annotation. All the other values in the G column have the unit `strike`; because that does not agree with the units for G10, there is an error

in the shaded cells in column H.

The type checker found only one spreadsheet with errors, the one containing the FREQUENCY formula problem which the unit checker also marked as having an error. We expected both checkers to flag this spreadsheet because the error comes from a misuse of the formula. The other two spreadsheets that the unit checker flagged passed the type checker because we did not rely on the wrong units present in the spreadsheet, but rather assigned the correct types for the value cells based on the header labels from the tables that they belonged to.

When testing our tool on existing spreadsheets, we found that the unit/type annotation process involved relatively little time and effort. All the correctly annotated spreadsheets passed the checkers; both checkers detected errors in those spreadsheets where we intentionally introduced them or where errors were present already. Our experience with these spreadsheets suggests that our approach offers promise to be a useful tool for real-world users of spreadsheets.

Chapter 6

Related Work

There are two research streams that feed into our unit checker work. Many researchers have suggested adding units directly to programming languages. We have already alluded to Kennedy's work on integrating units into ML [12] and a System F-like language [13]. Like these systems, XeLda uses unit polymorphism: the $+$ operator, for example, works with arbitrary units. Our inference algorithm does not rely on unification, as in Kennedy's ML system. Unlike ML-based languages, in which unification restricts mapping operators to work on homogeneous aggregates, the mapping operator in XeLda used to construct tables, is polymorphic over the possibly-heterogeneous units of elements in aggregates. Also, XeLda only works with a fixed set of functions, just those provided by Excel, not user-defined functions. Around the same time as Kennedy was developing his ML system, Goubault also proposed a dimension system for ML, using a sorted type algebra [9], allowing rational dimension exponents where Kennedy requires integers. As described, our system requires integer exponents. In a brief article, Wand and O'Keefe proposed integrating units with the simply-typed lambda calculus [21]. In their system, the basic units are fixed, although a programmer can add new units within a delimited scope.

There have also been some earlier proposals for adding units to programming languages, most notably the work of House [10]. More recently, Rittri considered dimensional analysis in the presence of polymorphic recursion [19].

Along with the work on units in programming languages, there has been some work on detecting errors in spreadsheets. The most closely related work to our type checker is the checker of Erwig and Burnett [4]. Their system is based on the same principles as ours. There is, however, a significant difference between their formulation and ours. Their system fails to distinguish between the *is-a* and *has-a* relationship. Additionally our system is capable of handling subtraction in cell expressions. These changes lead to a more thorough type system. We present the following examples to highlight valid spreadsheets that would be allowed to pass unhindered by our system, yet fail to pass through the system in [4].

Consider Figure 2.3. Under the system in [4], cells B3 and B9 would have the types `Gross Sales[TVs]` and `Costs[TVs]` respectively. In cell B15, we are subtracting the cost of TVs from the gross sales of TVs, to obtain the profits. Applying our rules,

the checking progresses as follows. Cells B3 and B9 are given the types $\text{Electronics[TVs]\{Gross Sales\}}$ and $\text{Electronics[TVs]\{Costs\}}$. Cell B15 requires checking the following operation:

$$\text{Electronics[TVs]\{Gross Sales\}} + \text{Electronics[TVs]\{Cost\}} = \text{Electronics[TVs]\{Gross Sales} \circ \text{Cost\}}$$

In this case it is not possible to apply Erwig and Burnett's rules, since they do not include a type transformation for the subtraction operation. Hence, their system would mark this as an error.

In Figure 2.4, we rearrange the tables in Figure 2.3, and assume that the header inference is able to infer that TVs and VCRs are both types of electronic goods. Again, 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 types $\text{All Electronics[TVs]\{Gross\}}$ and $\text{All Electronics[VCRs]\{Gross\}}$ respectively. The subsequent addition operation in cell B15 fails, because the hierarchies of the two types 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 types. Cells B3 and B9 could be assigned types Gross[TVs] and Gross[VCRs] , enabling cell B15 to pass the checker. 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:

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

Cell D3 is:

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

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. These failures were highlighted while we implemented their unit checker, which Erwig and Burnett lacked.

There have been other works tackling the specific problem of detecting errors in spreadsheets. Rothermel et al. [22] 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 [25], through the use of data flow adequacy criteria. Specifically the authors define the data flow test adequacy criteria employed, in terms of definition-use (du) associations that are involved in visible cell outputs. With the use of user interaction to validate the values in cells, the system marks du-associations as having been exercised, and visually reflect the percentage of all du-associations exercised per cell with shades of colors. Spreadsheet testing forms a component of the underlying spreadsheet systems we are checking. Rothermel et al.

apply this kind of testing to the Forms/3 spreadsheet language [23], 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 2.1, suppose the cell B5 contained the cell expression B3 + C4. Our type system would flag an error due to the addition of the types:

Electronics[TVs]{Quantity} & Year[2001] + Electronics[VCRs]{Quantity} &
Year[2002]

However the system in [22] would not be able to detect this problem in Figure 2.1.

Peyton-Jones, Blackwell, and Burnett have designed a language for adding *user-defined* functions into Excel [16]; adding unit-checking might require a different design than XeLda now uses. They propose adding matrices as first-class values, which could obviate the special treatment XeLda now uses. Because Excel uses ad hoc methods for assigning values to cells, some of our analysis techniques have to follow suit. Perhaps a more uniform spreadsheet language, such as proposed by Peyton-Jones et al. would lead to a cleaner design for a unit-checker.

There has been some work on data visualisation in spreadsheets. Igarashi et al. [24] have designed a system to visualize dataflow in spreadsheets through transient local views and static global views. They provide a semantic navigation through the spreadsheet with the possibility of modifications through graphical editing techniques. We could incorporate these techniques into our tool to enhance the error reporting and ease the user interaction with the spreadsheet.

MysterX, the COM scripting extension for PLT Scheme used to construct our tool is described in [20]. There are COM bindings for other functional languages, in particular, for Haskell [14]. Pucella has described a module system for COM [17] and formalized certain aspects of COM [18].

Chapter 7

From Prototype to Product

We have designed and implemented a unit/type-checker for Microsoft Excel that is able to handle its complex idioms. Our tool was able to find errors in off-the-shelf spreadsheets, validating our effort. We believe that this prototype also justifies the combination of functional programming and component programming. Although COM was designed with more conventional languages in mind, it works well with functional languages, including Scheme.

At this point, our tool is a prototype. To turn it into a true product, we will need several kinds of improvements:

- We need to design unit transformers and constraint generators for all Excel functions. There are over 300 such functions in Excel 2002, although many of those do not operate on numbers. One class of interesting Excel functions operate on relational databases embedded in spreadsheets. For example, the `DAVERAGE` function takes a range of cells representing the data, a column name, and another range that specifies query-by-example criteria; it returns the numeric average of the cells meeting the criteria. Hence, all cells in the named column should have the same units. There are also some odd cases. For example, one can apply `AND` to numbers as well as boolean values.
- We need feedback from experienced Excel users about how the application interface might be better designed. Instead of launching Excel from our application, for instance, it might be nicer if the application were launched from Excel.
- Currently, the user interface uses `MrEd` graphical classes, which use a fair amount of memory. As an alternative, `MysterX` can also use Internet Explorer for graphics, which might require fewer memory resources.
- We should add features based on properties of units. We could create an internal table of commonly-used units, allowing automatic conversions between unit systems. Given a spreadsheet with a worksheet with values in English units, for example, we could automatically create a new worksheet with values in SI units.
- We should integrate the unit checking and type checking aspect of `XeLda`. The user should be able to select through the interface the kind of checking to be performed.

- We should work on the header inference problem so that the units will be annotated and the types inferred, allowing for both checkers to work on the same spreadsheet, where the type annotations should be implicitly derived through the inference mechanism and the unit annotations should be explicitly entered by the user.

Although automatic unit conversion seems straightforward, there are a few subtleties that have kept us from implementing this feature. Consider a spreadsheet with the following values in cells: A1 containing the number 5 and annotated with unit `inches`, B1 containing the number 3 and annotated with unit `centimeters`, and C1 containing the formula `A1+B1`. Because Excel has no knowledge of units, it will always compute 8 as the result in C1 even though this answer is clearly incorrect.

In order to fix this problem, the unit checker has to modify the Excel formula in C1 by multiplying one of the operands by a conversion factor: for instance, multiplying A1 by 2.54 to convert it to `centimeters` so these can be added to the value in B1. In turn, if C1 has a unit annotation, the result has to be converted from the unit of the result to the annotated unit.

In short, supporting physical units takes more than just providing conversion rules between units; it requires rewriting the formulae, both to make the inputs consistent and to present the results in the units the user prefers. These rules also slightly complicate the process of determining unit errors, because an exact match is no longer necessary. Finally, while some units (such as physical units) have well-defined conversion factors, other “units” do not: it is reasonable to add dollars and euros, for instance, but the conversion factor literally changes constantly.

The most problematic issue we currently have with our tool is the speed of access to Excel through COM, which makes checking large spreadsheets time-consuming. Probably the most direct solution to this problem is to modify the MysterX extension to use COM Direct Interfaces, rather than COM Automation.

Bibliography

- [1] JPL Mars program. Mars Climate Orbiter Failure Board releases report, numerous NASA actions underway in response. Press release, Nov. 1999. Release 99-134, available at:
<http://mars.jpl.nasa.gov/msp98/news/mco991110.html>.
- [2] M. Burnett, C. Cook, O. Pendse, G. Rothermel, J. Summet, and C. Wallace. End-user software engineering with assertions in the spreadsheet paradigm. In *25th Intl. Conf. on Software Engineering*, 2003.
- [3] M. Burnett and M. Erwig. Visually customizing inference rules about apples and oranges. In *2nd IEEE Int. Symp. on Human-Centric Computing Languages and Environments*, pages 140–148, 2002.
- [4] M. Erwig and M. Burnett. Adding apples and oranges. In *4th Intl. Symp. on Practical Aspects of Declarative Programming*, volume 2257, pages 173–191. Springer-Verlag, 2002.
- [5] G. Filby, editor. *Spreadsheets in Science and Engineering*. Springer, 1995.
- [6] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, March 2002.
- [7] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching bugs in the web of program invariants. In *Proc. ACM SIGPLAN Conf. on Programming language design and implementation (PLDI '96)*, pages 23–32, 1996.
- [8] M. Flatt, R. B. Findler, S. Krishnamurthi, and M. Felleisen. Programming languages as operating systems (or revenge of the son of the Lisp machine). In *Intl. Conf. on Functional Programming (ICFP '99)*, pages 138–147, 1999.
- [9] J. Goubault. Inférence d'unités physiques en ML. *Journées Francophones des Langues Applicatifs*, pages 3–20, 1994.
- [10] R. House. A proposal for an extended form of type checking of expressions. *Computer Journal*, 26(4):366–374, 1983.

- [11] A. Kennedy. Dimension types. In D. Sannella, editor, *5th European Symp. on Programming (ESOP'94)*, volume 788 of *Lecture Notes in Computer Science*, pages 348–362. Springer, 1994.
- [12] A. Kennedy. *Programming Languages and Dimensions*. PhD thesis, University of Cambridge, April 1996.
- [13] A. Kennedy. Relational parametricity and units of measure. In *ACM Symp. on Principles of Programming Languages*, pages 442–455, 1997.
- [14] D. Leijen, E. Meijer, and J. Hook. Haskell as an Automation controller. In *Advanced Functional Programming, 3rd Internatl. School*, volume 1608 of *Lecture Notes in Computer Science*, pages 268–289. Springer, 1999.
- [15] R. R. Panko. What we know about spreadsheet errors. *J. End User Computing*, 10(2):15–21, Spring 1998.
- [16] S. Peyton-Jones, A. Blackwell, and M. Burnett. Improving the world's most popular functional language: user-defined functions in Excel. Unpublished article, Nov. 2002. Available from:
<http://research.microsoft.com/user/simonpj/papers/excel/>.
- [17] R. Pucella. The design of a COM-oriented module system. In *Proc. Joint Modular Languages Conference*, pages 104–118, 2000.
- [18] R. Pucella. Towards a formalization for COM, Part I: The primitive calculus. In *Proc. ACM SIGPLAN Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '02)*, pages 331–342, 2001.
- [19] M. Rittri. Dimension inference under polymorphic recursion. In *Functional Programming and Computer Architecture (FPCA '95)*, pages 147–159, 1995.
- [20] P. A. Steckler. Component support in PLT Scheme. *Software-Practice and Experience*, 32:933–954, 2002.
- [21] M. Wand and P. O'Keefe. Automatic dimensional inference. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: in honor of J. Alan Robinson*, pages 479–486. MIT Press, 1991.
- [22] G. Rothmel, M. Burnett, L. Li, C. Dupuis, A. Sheretov. A methodology for testing spreadsheets. In *ACM Transactions on Software Engineering and Methodology*, pages 110–147. ACM Press, 2001.
- [23] M. Burnett, J. Atwood, R. Djang, H. Gottfried, J. Rechwein, S. Yang. Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm. In *Journal of Functional Programming*, pages 155–206, 2001.
- [24] T. Igarashi, J. D. Mackinlay, B. W. Chang, P. T. Zellweger. Fluid Visualization for Spreadsheet Structures. In *14th IEEE Symposium on Visual Languages*, pages 118–125, Sept., 1998.

- [25] R. Panko and R. Halverson. Spreadsheets on trial: A survey of research on spreadsheet risks. In *Twenty-Ninth Hawaii International Conference on System Science*, 1996.
- [26] 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, 1999.
- [27] B. C. Pierce. *Types and Programming Languages*, MIT Press, 2002
- [28] R. Panko. Finding Spreadsheet Errors In *InformationWeek*, May, 1995
- [29] C. Fellbaum. *WordNet: And Electronic Lexical Database*, MIT Press, 1998
- [30] T. Knotzer. Startup Adds Collaboration Features to Excel
<http://www.informationweek.com/story/IWK20020325S0004>
- [31] E. Colkin. Nasdaq Giving XBRL a Try. In *InformationWeek*, Aug., 2002.
- [32] A. Ricardela and J. Maselli. To The Middle: Big ERP vendors haven’t doen well in the midmarket. Can Microsoft do better? In *Information Week*, May, 2002.
- [33] A. I. Katz. Academic Computing In U.S. Colleges And Universities: A Survey In *Journal of Information Systems Education*, vol. 4, Dec., 1992.
- [34] Microsoft Corporation. *Microsoft Component Object Model*.
<http://www.microsoft.com/com>

Appendix

Unit inference rules

Let:

- $\nabla \equiv$ any binary operator
- $\nabla^* \equiv$ any binary operator except for $+/-$
- $I(d)$ are the *is-a* headers for header d (possibly \emptyset)
- $U(d)$ is the unit for header d
- $\mathcal{I}(a)$ are the *is-a* headers for the cell at location a
- $\mathcal{U}(a)$ is the unit for the cell at location a
- $d \rightarrow h$ shows header d *has-a* header h
- $v(a)$ is the value of the cell at location a .
- \vec{u} ($= u_1[u_2[\dots[u_n]\dots]]$) is the short-hand representation for a hierarchy of *is-a* relationships.
- if $\vec{u} = u_1[\dots[u_n]\dots]$ then $\vec{u}[u'] = u_1[\dots[u_n[u']]\dots]$

Unit construction rules:

Headers:

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

Values:

$$\frac{\vdash d \in I(a) \quad \vdash U(d) = u \quad \vdash U(d)[p] = \{u\}}{\vdash d \in I(a) \quad \vdash U(d) = u \quad \vdash U(d)[p] = \{u\}}$$

$$\vdash \{d, d_1, \dots, d_n\} = \mathcal{I}(a), \text{ , } \forall i \in 1 \dots n : U(d_i) = u_i$$

References:

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

\oplus -rule :

$$\frac{\vdash u_1 = c_1[\dots[c_1[x_1 \dots [x_r[c_{i+1} \dots [c_j[\dots]]]]]] \dots] \quad \vdash u_2 = c_2[\dots[c_2[y_1 \dots [y_l[c_{i+1} \dots [c_j[\dots]]]]]] \dots] \quad \vdash u_1 \quad \vdash u_2 \quad i > 0; j \geq i; k, l \geq 0}{\vdash u_1 \oplus u_2 \leftarrow c_1[\dots[c_1[\dots[c_j[\dots]]]] \dots]}$$

$\&$ -rule:

$$\frac{\vdash u = n[\dots[u_i[\dots]]] \quad \vdash v = v[\dots[v_j[\dots]]] \quad \vdash u \neq v}{\vdash u \& v}$$

Simplification rules:

$$\begin{aligned} u_1 \& (u_2 \& u_3) &= (u_1 \& u_2) \& u_3 \\ u_1 \& u_2 &= u_2 \& u_1 \\ \bar{u} \& u &= \bar{u} \\ u_1 \& (u_2 \Delta u_3) &= (u_1 \& u_2) \Delta u_3 \end{aligned}$$

Identical *is-a* rule:

$$\frac{\vdash \bar{u}\{h_1\} \quad \vdash \bar{u}\{h_2\}}{\vdash \bar{u}\{h_1\} \Delta \bar{u}\{h_2\} \leftarrow \bar{u}\{h_1 \circ h_2\}}$$

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

$$\frac{\vdash \bar{u}_1\{h\} \quad \vdash \bar{u}_2\{h\}}{\vdash \bar{u}_1\{h\} \oplus \bar{u}_2\{h\} \leftarrow \bar{u}_1\{h\} \oplus \bar{u}_2\{h\}}$$

$$\frac{\vdash \bar{u}_1\{h\} \quad \vdash \bar{u}_2\{h\}}{\vdash \bar{u}_1\{h\} \Delta \bar{u}_2\{h\} \leftarrow \bar{u}_1\{h\} \Delta \bar{u}_2\{h\}}$$