Visually Customizing Inference Rules About Apples and Oranges

Margaret Burnett* and Martin Erwig
Department of Computer Science
Oregon State University
Corvallis, OR 97331, USA
[burnett|erwig]@cs.orst.edu

Abstract

We have been working on a unit system for end-user spreadsheets that is based on the concrete notion of units instead of the abstract concept of types. In previous work, we defined such a system formally. In this paper, we describe a visual system to support the formal reasoning in two ways. First, it supports communicating and explaining the unit inference process to users. Second and more important, our approach allows users to change the system's reasoning by adding and customizing the system's inference rules.

1 Introduction

Static type checking helps to find programming errors early, which makes programs more reliable and predictable. However, static typing has seldom been used in end-user programming languages. A possible reason for this omission is that the introduction of a type system incurs learning cost: either the cost of learning about type declarations, or the cost of understanding a type inference system well enough to understand the error messages it generates. End users are not usually interested in paying these costs.

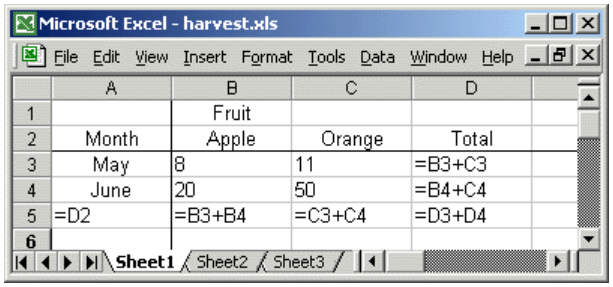
To help prevent some kinds of spreadsheet errors, we are developing an approach to reasoning about units. The work has some similarity to type inference [11, 3] in that it reasons behind the scenes through static analysis techniques to find errors in combining values, but our approach is not based on types. In particular, we are not translating abstract type systems into visual languages as exercised, for example, in [14, 18]. Rather, like other research into units and dimensions [24, 9], the goal of our approach is to detect errors related to illegal combinations of units. However, unlike the other works on units, we aim to detect any such error as soon as it is typed in, to make use of information such as column headers the end user has entered for reasons other than unit inference, and to support a kind of polymorphism of units through generalization. Note that our notion of "unit" is completely application dependent and is generally not related to the idea that units represent scales of measurement for certain dimensions [10].

We explain the ideas behind our approach by an example. Suppose a user has created the spreadsheet that is shown in Figure 1. From the labels, values, formulas, and their relative positions, the system can guess that, for example, the entries of column B are apples. The system confirms its guesses by interacting with the user, so that the user can correct the guesses and add additional information about the units structure as well. This mechanism for getting explicit information about units is a "gentle slope" language feature [13, 12]: the user does not have to "declare" any unit information at all, but the more such information the user enters through column headers or later clarifications in correcting the system's guesses, the more the system can use this information to reason about errors.

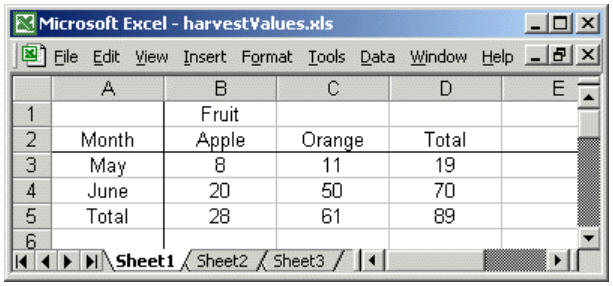
The system infers that the unit for the cell B5 is apples since the formula adds two numbers which are of unit apple. The entry in D3 adds apples and oranges, which at first glance may seem an illegal combination of units; however, it represents the total of all of row 3, which is in units of May as well as in units of all the fruits. Thus, the total is in units of May apples or May oranges, which reduces to May fruits, and is legal as well. As this demonstrates, in cells such as B3 there is a collaborative relationship between two kinds of units: apples and May. By similar reasoning, the total in D5 is legal; it turns out to be the sum of all fruits in all months, and its units reflect a collaborative relationship between fruits and months.

Now if the user attempts to add May apples to June oranges (B3+C4), the system immediately detects a unit error in this formula, because there is no match on specific units (apples versus oranges), and not enough is being combined to cover all fruits in a way that also matches either

*This work was supported in part by the National Science Foundation under ITR-0082265.



(a) Formulas



(b) Resulting values

Figure 1. A fruit production spreadsheet.

one month or generalizes to all months. Adding May apples to June oranges is the kind of error that arises when a user accidentally refers to the wrong cell in a formula, through typographical error or selecting the wrong cell with the mouse.

In previous work [5], we developed a formal reasoning system for detecting such errors. However, the reasoning system we developed was expressed in a highly formal notation that would be inappropriate for an audience of end users. Further, our reasoning system did not address how the system guesses about the labels so as to derive the hierarchies of units, how a user can correct the system when it guesses wrong, or how such a system can communicate its reasoning to the user. In this paper, we address these points. The implications of addressing these points go beyond matters of representation: they change the reasoning system in a fundamental way, namely by allowing the user to customize the system's inference rules themselves, not just the terms used in the system's inference rules.

This approach is, to the best of our knowledge, the first to support the following features:

- It supports not only communicating visually with the user about the results of static analysis, but also *customization* visually by the user of the static reasoning system.
- It is intended for *end-user* programmers.

2 Background: Reasoning about Units

In this section, we summarize the reasoning system we previously devised for units. More details, in particular, the formal definitions, can be found in [5].

The design of the unit system was driven by four goals that acknowledge research findings about how end users tend to work: First, the reasoning mechanism should give immediate visual feedback as to the unit safety of the most recently entered spreadsheet formula as soon as it is entered. Thus, we wanted the reasoning system to work *incrementally*. Second, the reasoning mechanisms should directly be in terms of elements with which the user is working, such as labels and operation names, that is, we were aiming at *directness*. Third, as a consequence of the third goal, the reasoning mechanism should require *no formal notion of types* per se other than what is expressed by units. Fourth, the reasoning mechanism should support the kind of spreadsheets end users really build. In particular, the approach should not rest on assumptions that end users will create "the right kind" of formulas, be complete in their labeling practices, or that their spreadsheets will be free of statically detectable errors. This should make the reasoning system *practical*.

2.1 What are Units?

The unit information for the cells in a spreadsheet are completely contained in the spreadsheet itself because units are defined by values. More precisely, each value in a spreadsheet (except blanks) defines a unit. Although all values are units, not all values are generally used as units. For example, in the spreadsheet from Figure 1 the text Total is by definition a unit, but it does not have practical use as a unit for cells in the spreadsheet.

The fact that one value is a unit for some cells in a spreadsheet is given by *header definitions*. Intuitively, a header is a label that gives a unit for a group of cells. For example, in Figure 1 Month is a header for the cells containing May and June.

Since units are values, they can themselves have units; hence, we can get chains of units called *dependent units*. Further, since values in a spreadsheet can be classified according to different categories at the same time, values can principally have more than one unit, which leads to *and units*. Finally, operations in a spreadsheet combine values that possibly have different units. In some cases, these different units indicate a unit error, but in other cases the unit information can be generalized to a common "superunit". Such generalizations are expressed by *or units*. In the following, we will explain these different forms of units in more detail.

Dependent Units. In the spreadsheet from Figure 1 we have that cell B2 (containing Apple) is a header for the cell

B3 (containing the number 8). On the other hand, B2 has itself cell B1 (Fruit) as a header. This hierarchical structure is reflected in our definition of units. In this example, the unit of the cell B3 is not just Apple, but Fruit[Apple]. In general, if a cell c has a value v as a unit which itself has unit u, then c's unit is a *dependent unit* $u[v]$. Dependent units are not limited to two levels. For example, if we distinguished red and green apples, a cell containing Green would have unit Fruit[Apple], and a cell whose header is Green would have the dependent unit Fruit[Apple][Green], which is the same as Fruit[Apple[Green]].

And Units. Cells might have more than one unit. For example, the number 11 in cell C3 gives a number of oranges, but at the same time describes a number that is associated with the month May. Cases like this are modeled with *and* units, which are similar to intersection types [17]. In our example, C3 has the unit Fruit[Orange]&Month[May].

Or Units. The dual to *and* units are *or* units that correspond to union types. *Or* units are inferred for cells that contain operations combining cells of different, but related units. For example, cell D3's formula is B3 + C3. Although the units of B3 and C3 are not identical, they differ only in one part of their *and* unit, Fruit[Apple] and Fruit[Orange]. Moreover, these units differ only in the innermost part of their dependent units. In other words, they share a common prefix that includes the complete path of the dependency graph except the first node. This fact makes the + operation applicable. The unit of D3 is then given as an *or* unit of the units of B3 and C3, that is, Fruit[Apple]&Month[May]|Fruit[Orange]&Month[May]. In general, an *or* unit is valid only if it can be transformed into a unit expression in which *or* is applied only to values (that is, not unit expressions) that all have the same unit.

Finally, for completeness of the formal unit system we have included a unit **1** that is used for all values that do not have specifically assigned any other unit and an error unit ε that represents unit errors (as well as erroneous computations).

2.2 Well-Formed Units

We consider only a subset of all possible unit expressions to be meaningful. For example, a number cannot represent a number of apples *and* oranges at the same time. Hence, a unit Apple&Orange is not valid. However, a number can well represent either apples *or* oranges. This is why we consider a unit like Apple|Orange to be valid.

By defining which unit expressions are valid we can distinguish between unit-correct computations and unit-errors. Formally, the concept of valid unit expressions is captured *well-formed units*. Five rules are needed to define when a unit is well formed:

1. **1** is always a well-formed unit.
2. Every value that does not have a header is a well-formed unit. For example, in Figure 1, Fruit is a well-formed unit.
3. If a cell has value v and header u, then $u[v]$ is a well-formed unit. For example, in Figure 1, Fruit[Apple] is a well-formed unit.
4. Where there is no common header ancestor, it is legal to *and* units. For example, in Figure 1, Fruit[Apple]&Month[May] is a well-formed unit because Apple and May have no common ancestor.
5. Where there is a common header ancestor, it is legal to *or* units. For example, in Figure 1, Fruit[Apple]|Fruit[Orange], which denotes the same unit as Fruit[Apple|Orange], is well-formed. More precisely, we require that all the values except the most nested ones agree. This is the reason why the unit Fruit[Apple[Green]]|Fruit[Orange] is not well-formed.

2.3 Unit Inference

It remains to be explained how unit expressions are derived for all cells of a spreadsheet. The main rules for inferring units for cells are given below.

1. All cells that do not have a header, have the unit **1**. For example, cell Total has unit **1**.
2. If cell a has a header cell b that contains a value v and has a well-formed unit u, then a's unit is $u[v]$. More generally, if a cell has multiple headers with values v_i and units u_i, then its unit is given by the *and* unit of all the $u_i[v_i]$. An example is the cell B3 containing 8 whose unit is Fruit[Apple]&Month[May].
3. If cell a's formula is a reference to cell b, then b's unit, say u_b is propagated to a. For example, cell A5 contains a reference to D2 which has unit **1**. Hence, A5's unit is also **1**. If a has itself a header definition, say u_a, then u_a must conform with u_b, which is achieved by defining a's unit to be $u_a \& u_b$.
4. Each operator has its own definition of how the units of its parameters combine.

Regarding the latter point, we define a function μ_ω for each operator ω, which defines how the operation transforms the units of its parameters. Note that the definition of μ_ω takes also into account the unit that can be derived from a possible header definition for that cell. Both sources of unit information have to be unified. This unification is particularly helpful to retain unit information in the case of multiplication and division because these two operations have in our current model only a weak unit support.

The definition of μ_ω is shown for some operations in Figure 2. u is the cell's header unit; $u_1, \ldots, u_n$ are all the units of the parameters.

$$\mu_+(u,u_1,\ldots,u_n) = (u_1|\ldots|u_n)\&u$$
$$\mu_{\mathsf{count}}(u,u_1,\ldots,u_n) = (u_1|\ldots|u_n)\&u$$
$$\mu_*(u,u_1,\ldots,u_n) = \downarrow(u_1,\ldots,u_n)\&u$$
$$\ldots$$
$$\downarrow(u_1,\ldots,u_n) = \begin{cases} u_i & \text{if } u_i \neq \mathbf{1} \wedge \forall j \neq i : u_j = \mathbf{1} \\ \mathbf{1} & \text{otherwise} \end{cases}$$

Figure 2. Unit transformations.

A proper treatment of multiplication and division by the reasoning system requires the concept of dimensions [24, 9]. Extending our unit system by dimensions would complicate it considerably; in particular, end users would probably be confused if required to cope with both unit and dimension error messages. However, the extensions described in this paper make it possible for users to supplement the system's knowledge with additional inference rules that could handle such cases.

By applying operations we can obtain arbitrarily complex unit expressions that do not always meet the conditions for well-formed units. We need equations on unit expressions that allow us to simplify complex unit expressions. Whenever simplification to a well-formed unit is possible, it can be concluded that the operation is applied in a "unit-correct" way. Otherwise, a unit error is detected. Simplification rules for doing so include rules for commutativity, generalization, factoring, and so on. The complete set of equations is given in Figure 3. They define a semantic equality of units.

$$\begin{aligned}
u_1\&u_2 &=_u u_2\&u_1 && \text{(commutativity)}\\
u_1|u_2 &=_u u_2|u_1 &&\\
(u_1\&u_2)\&u_3 &=_u u_1\&(u_2\&u_3) && \text{(associativity)}\\
(u_1|u_2)|u_3 &=_u u_1|(u_2|u_3) &&\\
u\&(u_1|u_2) &=_u (u\&u_1)|(u\&u_2) && \text{(distributivity)}\\
u\&u &=_u u && \text{(idempotency)}\\
u|u &=_u u &&\\
\mathbf{1}\&u &=_u u && \text{(unit)}\\
u[u_1]|u[u_2] &=_u u[u_1|u_2] && \text{(factorization)}\\
u[u_1|\ldots|u_k] &=_u u \quad \Leftarrow (*) && \text{(generalization)}\\
(u_1[u_2])[u_3] &=_u u_1[u_2[u_3]] && \text{(linearization)}
\end{aligned}$$

Figure 3. Unit equality.

The condition $(*)$ for generalization is that the *or* unit expression consists exactly of all units $u_1,\ldots,u_n$ that have u as a header. Note that & distributes over |, but not vice versa, and that although **1** is the unit for &, it leads to a non-valid unit when combined with |.

We will explain some of the inference rules and equality rules in the next section.

3 Visually Customizing Inference Rules

Consider the system's inferences for the spreadsheet of Figure 1. The system's inference rules at this point are those given in Figures 2 and 3. These figures do not give a way for the system to infer headers, so the system cannot yet do so. This means all values have unit **1**, and thus there are no unit errors.

To get the use of units into this picture, the system needs some way to get beyond this point. Thus, we provide a very simple unit inference rule based on spatial reasoning:

> *Base Layout Rule*: A cell's headers are the nearest cell to the left whose formula is a constant string and the nearest cell above whose formula is a constant string.

For example, cell B3's headers are A3 (May) and B2 (Apple), and B4's headers are A4 (June) and B2 (Apple). Not all units will come out so well using this rule, however. For example, according to it, A4's (June's) header is A3 (May). The user will help solve this kind of problem.

It should be noted that a much more sophisticated Base Layout Rule, or even a group of sophisticated Base Layout Rules based upon spatial reasoning, improved over time by noticing the particular user's habits, and so on, would be possible. The choice of Base Layout Rules is up to the environment's designer. Thus, our Base Layout Rule should be viewed as simply a prototype we have devised to help show how customization can proceed, not as a recommendation for any particular base rule.

Because of the Base Layout Rule, the system immediately detects some unit errors. For example, cell B5 has an error because its formula and spatial position leads to a unit Fruit[Apple]&Month[May]|Fruit[Apple]&May[June], which is equal to Fruit[Apple]&(Month[May]|May[June]) due to the distributivity law. However, this unit is not well-formed because Month[May] and May[June] do not share a common prefix and therefore cannot be simplified further.

When the system detects unit errors such as the one just described for B5, it displays an indicator of a unit error, currently planned as changing the cell's background to red. The user is able to obtain a visual explanation of the problem using the visual representation of the derivation process. (We will briefly describe this representation in Section 4.) From the explanation, the user can see that the system has not figured out all the headers correctly. The user can then customize the system's inference process to solve this problem by adding new rules.

3.1 Entering a New Rule by Demonstration

New rules are entered using a visual by-demonstration approach. We selected a visual by-demonstration approach

to expressing rules because the Kidsim/Cocoa/Stagecast [2, 6] and the AgentSheets [21, 7] projects have contributed empirical evidence that this approach can be used by at least two populations of end users, namely children and teachers (for example, [19, 20, 22]).

For example, suppose the user's viewing of the explanation shows the system's use of the Base Layout Rule in inferring headers. The user sees that this is a problem, and adds a new rule, shown in Figure 4.

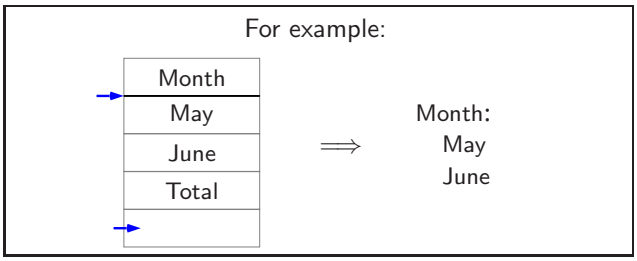


Figure 4. A layout rule given by the user.

As in Stagecast, the rule format is the condition, an implication arrow, and the result. Also as in Stagecast, the rule is based on visual attributes. For the "if" side, the user has snapped a picture of a portion of the spreadsheet. However, unlike in Stagecast, the system has added tiny arrows to the visual parts of the picture that are important other than the spatial layout between non-blank cells. In this example, the additional important parts are the presence of a line under Month and the presence of a blank after Total. The user can add, delete, and/or move these "importance arrows" as desired. The spatial layout between non-blank cells is always taken into account in pattern matching, and thus does not need importance arrows.

Thus, the "if" side of the picture says that when there is a vertical list with a line under one of the words, the list starts at the underlined word and ends at a blank. The "then" side of the picture starts as an automatically generated suggestion by the system, but the user can manipulate it as needed. In the presented example, the system has originally included Total, but the user has deleted it resulting in the "then" part actually shown in Figure 4. This "then" side defines a unit hierarchy whose parent (that is, header) is the underlined word and whose children are every element except the last one. The "For example" label indicates that the reasoning system does not care about the actual values of the cells, that is, that these are just an example. In other words, the user has defined a generic rule.

Two questions arise in this connection: (i) how does the system generate a suggestion for the "then" side? and (ii) what is the precise meaning of a generic rule? To answer these questions, we first have to explain the meaning of the "if" part of a rule. In general, the "if" part can be considered as a pattern that consists of constant and variable parts. Importance arrows mark constants. These can be formatting symbols like the black underline or particular cell values like the blank cell in Figure 4. All other parts are interpreted as variables. These variables can be partitioned into groups of either one or more cells. In particular, all cells that are (transitively) adjacent and that are not separated by constants comprise one group. Now if the partition obtained in the described way consists of one single variable and one group of variables, the system generates the visual form of a header definition with the one variable as the header/unit for the set of variables in the group. This answers the first question.

The meaning of this rule is that for all parts of the spreadsheet that match the "if" side of the rule, a header definition according to the "then" side is created by substituting the actual variables from the spreadsheet for the variables in the pattern. In that process the number of variables in a group does not matter. If the user has edited the system-generated "then" part, for example, by deleting a variable, these modifications are interpreted "by position", that is, from the header/unit definition the first/last, second/second-but-last, and so on variable from the group is excluded. We are currently not considering rules whose "if" part specifies anything other than a 2-partition of a single variable and a group. Overlapping patterns are possible and generally lead to *and* units of the parts that are matched multiple times.

This new rule becomes the first layout rule the system will check, before it reverts to applying the Base Layout Rule. In general, as in Stagecast, a new rule always becomes the first rule the system tries to apply. As soon as the system finds an applicable rule, it applies it and does not consider other rules that possibly match. This rule application strategy ensures that the unit simplification process is deterministic.[1]

Note that the "then" side of a rule is a logical relationship, not a spatial layout. The demonstrational rule-based languages Stagecast and AgentSheets express only spatial relationships in their rule syntax (although it is possible to express state information as well through additional variables). A discussion of how logical relationships are expressed follows next.

[1] On the other hand, such a fixed reduction strategy might prevent in some cases a reduction to a well-formed unit, which could be achieved by applying rules in a different order. However, this behavior agrees with the definition of most other type systems, which have to report a type error in case of doubt for the sake of type soundness [11]. For our unit system, knowledge about the confluence [4] of the current rule system could yield additional information about the confidence in a reported unit error, that is, a unit error reported in a confluent rule system is definitely an error.

3.2 Visual Representation of Logical Relationships

Most end-user rule-based languages do not provide a way to enter strictly logical relationships in the rules. Instead, all relationships expressed in the main portion of the rule must be based on visual attributes such as topology or spatial relationships. (One exception is FAR [1], in which rules are expressed via formula subexpressions and cell references, in a multiparadigm spreadsheet-based approach.) The reasoning system presented here specifically focuses on logical relationships, but on deriving them and using them to detect errors, not to determine cell values or to define a program per se. Hence, it produces a static analysis reasoning *about* a visual program (spreadsheet), not a program itself.

The user participates in how the reasoning can be done by helping to define the rules followed in doing this reasoning, and hence must be given access to the logical relationships the system infers.

One such logical relationship is the hierarchical nesting of units. Recall that in the formal notation of the underlying system, nestings are displayed using open and closed square brackets. The visual representation of these logical nesting relationships is to display them as nested lists, in order to explicitly communicate the hierarchical reasoning of the system. An example of this representation is the "then" side of the rule in Figure 4.

In general, there is a 1:1 mapping to a visual representation from every formal symbol used by the underlying system; see Figure 5. It is important for every element of reasoning of the underlying system to be represented to the user, since the user actually participates in customizing the system's reasoning.

The representation elements were not chosen arbitrarily. In particular, the representation of &, |, and parentheses is based upon the results of Pane and Myers's work on language syntax elements that are problematic for end-user programmers [15]. Specifically, they recommend that the use of these textual elements be omitted, as end users do not interpret them to have the intended meanings. Following this recommendation, in our approach & is represented by a horizontal list, | as a vertical list, and parentheses by enclosing boxes. Pane and Myers themselves defined a language called HANDS [16] that follows their earlier recommendations. In HANDS, & and | are replaced by vertical and horizontal lists respectively. Moreover, nested boxes are used to represent nested parentheses in a manner similar to our approach.

3.3 Non-Generic Rules

The layout rules that now exist, the Base Layout Rule supplemented by the new rule of Figure 4, are still not enough to correctly derive the header for Orange, because the new rule handles only vertical lists, and the Base Layout Rule will decide Orange's header is Apple. The user can solve this problem by entering a new rule to handle horizontal lists in a manner similar to that described above. Alternatively, the user can provide specific information about the Fruit hierarchy by entering a specific rule about it, as in Figure 6. The main difference between a specific rule and a "for example" rule is that the "for example" label is missing.

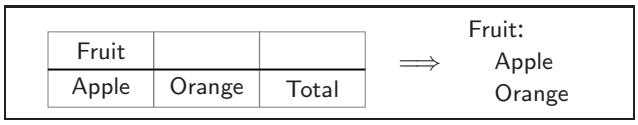


Figure 6. A non-generic rule defining the fruit hierarchy.

The left hand side of the fruit rule is not really necessary. If it were not present, all that would be left is the right hand side, which simply defines the fruit hierarchy. However, the left hand side does perform some extra safeguarding, because if the user eventually changes the spreadsheet by adding another column of fruit, the fruit rule will no longer fire for any of the fruits, so that unit errors will pop up all over the spreadsheet, providing a great deal of immediate feedback that the user's last edit has caused the reasoning about units to go wrong.

Because the user can enter specific, concrete rules such as the fruit rule, it is possible for the system to successfully reason about large spreadsheets with inconsistent layout conventions. Specific, concrete rules allow the user to let the system work around exceptions to his or her usual layout conventions. As the next section will explain, they also allow the user to customize the system's treatment of *and* units.

3.4 Meters * Meters = SquareMeters

As we previously mentioned in Section 2, our reasoning system does not reason about dimensions, such as multiplying meters times meters to get square meters. According to our inference rules, multiplying meters times meters will in result in the unit **1**; see the definition of μ_* in Figure 2.[2]

[2]The rationale behind this definition is to rather accept weaker unit transformations, here: weakening multiplications of non-**1** units to **1**, than to report unit errors that could not really be remedied within the static rule system without changing either the labels ot the header definitions of the spreadsheet; see [5].

Concept	Formal notation	Formal example	Visual representation	Visual example
and unit	$u\&u'$	May&Apple	$u\ u'$ (horizontal list)	May Apple
or unit	$u\|u'$	Apple|Orange	u / u' (vertical list)	Apple / Orange
unit definition		(assumed to be given)	$u:$ / u_1 / … / u_n (nested headers)	Fruit: / Apple / Orange
dependent unit	$u[u']$	Fruit[Apple]	u / u' (nested labels)	Fruit / Apple
brackets	(u)	Fruit&(May|June)	$\boxed{u}$ (box around expression)	Fruit [May / June]
one unit	**1**	**1**	(omitted)	
unit error	ε	ε	red cell background	B3+C4

Figure 5. Visual representation of unit expressions.

However, it is possible for the user to provide specific, concrete rules to solve this problem. This is done using the syntax shown in Figure 7. For the sake of concreteness, the overriding of rules involving *and* units is expressed by applying an operation like $*$ to units (and not a unit operator &). The meaning is that whenever multiplication is applied to values/cells whose unit is Meters, the resulting unit of the cell containing this formula is SquareMeters, that is, the default unit transformation for $*$ is superseded.

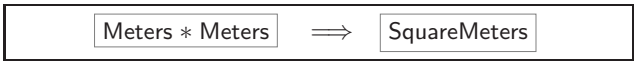


Figure 7. A "dimension" rule.

Once the SquareMeters unit has been so defined, it can be included in rules about hierarchical relationships such as those expressed in the right hand side of the fruit rule.

Combining value operations with units has the nice effect that we can express unit transformations *and* the association of these to operations in one definition. (Using, for example, the rule Meters Meters ⇒ SquareMeters would convert any unit expression Meters&Meters into SquareMeters regardless of the operation that created it, which is probably wrong.)

This technique allows the user to solve any unit error that the system has inferred because it did not have complete information. For example, it may be valid to add May apples to June oranges, because those are the only items in the Safeway grocery chain's order. The user can provide a rule such as the square meters rule that acknowledges SafewayItems as a valid unit.

Hence, the described form of (non-generic) overriding of the built-in rules for *and* units enables a limited treatment of dimensions within the unit system. This approach, however, does not provide a comprehensive treatment of dimensions. For example, if we had a formula in our spreadsheet that divided a cell with unit SquareMeters by a cell having unit Meters, the rule system cannot automatically infer that the cell should be of unit Meters. To realize this behavior, the user would have to define a further rule for division and meters. Also, in order to deal with square inches, square miles, and so on, the user had to enter each rule individually.

To solve the latter problem we could be tempted to allow generic versions of these kinds of rules for *and* units, simply by making them "for example" rules, but then we encounter two problems. First, we would have to address the fact that one unit like SquareMeters consists of a constant part (Square) and a variable part (Meters). We could follow a simple textual approach that matches all parts of the left hand side of the rule in the resulting unit and consider these parts as variable. However, this approach would not scale up very well to more complex unit expressions, in particular, for nested dimensions. Second, a completely generic version of the dimension rule would probably be too general because it would allow us, for instance, to multiply apples times apples—for the cell containing such a formula the inference system would then derive a unit SquareApple, which does not make any sense. Hence, if we wanted to allow "for example" rules for *and* units, we would need a way of restricting the set of units to which the general rule applies. This would amount to going from parametric polymorphism to a controlled form of overloading [23] or qualified types [8].

3.5 Rules About Rules

The unit approach for spreadsheets leads to each spreadsheet having its own customized unit system, because the units are built from the values contained in the spreadsheet. Moreover, the user can dynamically redefine the unit system by adding rules or unit definitions. Rules or definitions that can be added are:

1. *Generic hierarchy rules*. These are the "for example" rules like the one shown in Figure 4. These rules define a general mechanism to infer headers and dependent units in a spreadsheet. They do not really modify the reasoning system, but they feed the spatial and visual characteristics to it.
2. *Specific unit definitions*. These are rules like the one shown in Figure 6. These rules extend the unit hierarchy (once); they can be considered special cases of generic hierarchy rules, in that both are similar to definitions of implicit type definition schemas.
3. *Concrete unit transformations*. These are definitions of new units and how they are related to specific operations like the one shown in Figure 7. These rules actually modify the unit reasoning system because they override unit transformations for operations applied on cells with particular units.

A user can change or delete the rules he or she has added to a unit system of a spreadsheet, but it is not possible to delete or change built-in rules like the ones shown in Figure 3. However, by adding concrete unit transformations the behavior of the core reasoning system can be modified. In particular, since new rules are considered first, before the system-provided ones, users have a lot of power in the ways they can affect the reasoning system.

The described unit system actually defines a three-level reasoning system about spreadsheets. At level one, the built-in rules and the Base Layout Rule provide a general framework for reasoning about spreadsheets. Level two is provided by the values and their spatial arrangement of each spreadsheet. Together with level one, it provides a rudimentary unit checking system that is already partially customized for the application at hand. Finally, level three is given by the user-defined rules. This level can be used to customize the unit inference system further to any desired level of precision.

4 Current Status and Future Work

We have devised our representation with the end user in mind and have drawn from others' empirical work in choosing the syntax. Still, there is no substitute for empirical work on the end product with real people, and that is an important next step for us. Also, the implementation is still in a very rough prototype form. Specifically, we have implemented a research prototype of the reasoning system using Haskell. Our prototype includes the inference rules, but does not yet automatically generate the visual mechanisms shown in this paper.

One particularly important aspect of the visual interface is the explanation of unit errors. In addition to the "formula view" and "values view" of a spreadsheet as, for example, provided by Excel, we also offer the user a "units view". In the units view, a user sees the values in grey print, and superimposed on them in regular print are the units. (The faded grey values are just for context). The erroneous ones have a red background. Then the user can ask for an explanation. There is a static and a dynamic representation of the explanations. The static one gives a story board, and the dynamic one animates it by transitioning from frame to frame through the story board. The static one looks just like the rules we showed in this paper, only with the examples actually in the paper filled in. The system's inference rules also have a visual representation using the same syntax as in the paper, so they can be handled in this explanation in the same way the user ones are.

5 Conclusion

We have presented a visual approach to reasoning about units in spreadsheets. The approach is a "gentle slope" approach in the sense that the user does not have to learn anything new to start using it, but the more information he or she chooses to provide to the system, the more helpful the system can be in reasoning about whether the spreadsheet's different units are being combined correctly.

In particular, we have shown how the user can customize the reasoning system by visual rules. The user can employ rules to provide explicit information about particular unit hierarchies, to modify the inference behavior of the system in general, and to define unit transformation to cope with those special cases the system is not able to handle appropriately.

Altogether, the user has a three-level unit inference system available that can reason about units in spreadsheets with any desired degree of precision.

References

[1] M. M. Burnett, S. Chekka, and R. Pandey. FAR: An End-User Language to Support Cottage E-Services. In *1st IEEE Symp. on Human-Centric Computing Languages and Environments*, pages 195–202, 2001.

[2] A. Cypher and D. Smith. KidSim: End-User Programming of Simulations. In *ACM Conf. on Human Factors in Computing Systems*, 1995.

[3] L. Damas and R. Milner. Principal Type Schemes for Functional Programming Languages. In *9th ACM Symp. on Principles of Programming Languages*, pages 207–208, 1982.

[4] N. Dershowitz and D. A. Plaisted. Rewriting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, chapter 9. Elsevier Science Publishers, Amsterdam, NL, 2001.

[5] M. Erwig and M. M. Burnett. Adding Apples and Oranges. In *4th Int. Symp. on Practical Aspects of Declarative Languages*, LNCS 2257, pages 173–191, 2002.

[6] N. Heger, A. Cypher, and D. Smith. Cocoa at the Visual Programming Challenge 1997. *Journal of Visual Languages and Computing*, 9(2):151–169, 1998.

[7] A. Ioannidou and A. Repenning. End-User Programmable Simulations. *Dr. Dobb's Journal*, 1999(August):40–48, 1999.

[8] M. P. Jones. A Theory of Qualified Types. In *3rd European Symp. on Programming*, LNCS 582, pages 287–306, 1992.

[9] A. Kennedy. Dimension Types. In *5th European Symp. on Programming*, LNCS 788, pages 348–362, 1994.

[10] A. Kennedy. Relational Parametricity and Units of Measure. In *24th ACM Symp. on Principles of Programming Languages*, pages 442–455, 1997.

[11] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:248–375, 1978.

[12] B. Myers, S. Hudson, and R. Pausch. Past, Present, and Future of User Interface Software Tools. *ACM Transactions on Computer-Human Interaction*, 7(1):3–28, 2000.

[13] B. Myers, D. Smith, and B. Horn. Report of the 'End-User Programming' Working Group. In B. Myers, editor, *Languages for Developing User Interfaces*, pages 343–366. A. K. Peters, Ltd., Wellesley, MA, 1992.

[14] M. Najork. Programming in Three Dimensions. *Journal of Visual Languages and Computing*, 7(2):219–242.

[15] J. F. Pane and B. A. Myers. Tabular and Textual Methods for Selecting Objects from a Group. In *16th IEEE Symp. on Visual Languages*, pages 157–164, 2000.

[16] J. F. Pane and B. A. Myers. The Impact of Human-Centered Features on the Usability of a Programming System for Children. In *ACM Extended Abstracts: Conf. on Human Factors in Computing Systems*, 2002.

[17] B. C. Pierce. Intersection Types and Bounded Polymorphism. *Mathematical Structures in Computer Science*, 7(2):129–193, 1997.

[18] J. Poswig and C. Moraga. Incremental Type Systems and Implicit Parametric Overloading in Visual Languages. In *9th IEEE Symp. on Visual Languages*, pages 126–133, 1993.

[19] C. Rader, C. Brand, and C. Lewis. Degrees of Comprehension: Children's Understanding of a Visual Programming Environment. In *ACM Conf. on Human Factors in Computing Systems*, pages 351–358, 1997.

[20] C. Rader, G. Cherry, C. Brand, A. Repenning, and C. Lewis. Designing Mixed Textual and Iconic Programming Languages for Novice Users. In *14th IEEE Symp. on Visual Languages*, pages 187–194, 1998.

[21] A. Repenning and J. Ambach. Tactile Programming: A Unified Manipulation Paradigm Supporting Program Comprehension, Composition and Sharing. In *12th IEEE Symp. on Visual Languages*, pages 102–109, 1996.

[22] M. B. Rosson and C. D. Seals. Teachers as Simulation Programmers: Minimalist Learning and Reuse. In *ACM Conf. on Human Factors in Computing Systems*, pages 237–244, 2001.

[23] P. Wadler and S. Blott. How to Make Ad Hoc Polymorphism Less Ad Hoc. In *16th ACM Symp. on Principles of Programming Languages*, pages 60–76, 1989.

[24] M. Wand and P. O'Keefe. Automatic Dimensional Inference. In J.-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 479–483. MIT Press, Cambridge, MA, 1991.