Adding Apples and Oranges

Martin Erwig and Margaret Burnett

Oregon State University
Department of Computer Science
Corvallis, OR 97331, USA
[erwig|burnett]@cs.orst.edu

Abstract. We define a unit system for end-user spreadsheets that is based on the concrete notion of units instead of the abstract concept of types. Units are derived from header information given by spreadsheets. The unit system contains concepts, such as dependent units, multiple units, and unit generalization, that allow the classification of spreadsheet contents on a more fine-grained level than types do. Also, because communication with the end user happens only in terms of objects that are contained in the spreadsheet, our system does not require end users to learn new abstract concepts of type systems.

Keywords: First-Order Functional Language, Spreadsheet, Type Checking, Unit, End-User Programming

1 Introduction

The early detection of type errors is a well-known benefit of static typing, but static typing has not been used in programming languages intended for end users not formally schooled in programming. A possible reason for this omission is that the introduction of static types 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, because their use of programming is simply a means to an end, namely helping them get their "real" jobs done faster.

The number of end-user programmers in the United States alone are expected to reach 55 million by 2005, as compared to only 2.75 million professional programmers [4]. This estimate was originally made in 1994 based on Bureau of Labor Statistics and Bureau of Census figures, and a 1999 assessment of the model shows that its predictions are so far reasonably on track. The "programming" systems most widely used by end users are members of the spreadsheet paradigm. Henceforth, we use the term spreadsheet languages[1] to refer to all systems following the spreadsheet paradigm, in which computations are defined

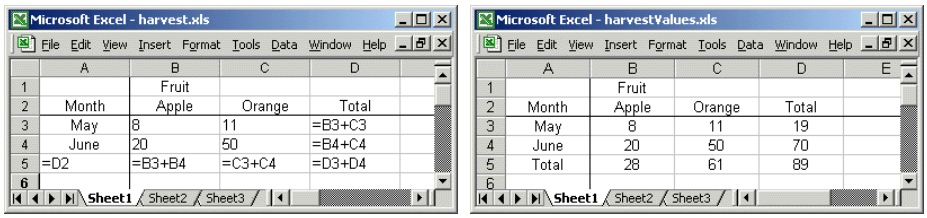
[1] We have chosen this terminology to emphasize the fact that even commercial spreadsheet systems are indeed languages for programming, although they differ in audience, application, and environment from traditional programming languages.

by cells and their formulas. Although spreadsheet languages have not been taken seriously by the programming language community, two recent NSF workshops' results included the conclusion that serious consideration of languages such as these is indeed needed [2].

There is extensive evidence that many spreadsheets contain errors. For example, field audits of real-world spreadsheets have found that 20% to 40% of all spreadsheets contain errors, and several controlled spreadsheet experiments have reported even higher error rates [3, 16, 14]. These errors can have serious consequences. For example, a Dallas oil and gas company lost millions of dollars in an acquisition deal because of spreadsheet errors [13].

To help prevent some of these spreadsheet errors, we are developing an approach to reasoning about units. Like other research into units and dimensions [17, 9], the goal of our approach is to detect errors related to illegal combinations of units. Unlike other works, 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].

Consider the following scenario. Suppose a user has created the example spreadsheet in Figure 1. From the labels the user has placed, the system can guess that, for example, the entries of column B are apples. The system confirms its guesses by interacting with the user, and 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 [12, 11]: 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.



(a) Formulas (b) Resulting values

Fig. 1. A fruit production spreadsheet.

From the information gleaned, the system can deduce that the total at the bottom of column B is also in apples, which is a legal combination of units. 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 suppose 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 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.

The rest of this paper is structured as follows. In Section 2 we discuss specific aspects of spreadsheets that influence the design of our unit system. In Section 3 we define a small language that serves as a model of spreadsheets. In Section 4 we develop a notion of units and unit expressions that can be used to describe units for expressions and cells. In particular, we have to describe what well-formed units are. Then in Section 5 we describe the process of checking the units of (cells in) a spreadsheet. We draw some conclusions and give remarks on future work in Section 6. Formal definitions appear in the Appendices A to C.

2 Features of Spreadsheet Languages that Impact Reasoning about Units

Like other members of the applicative family, spreadsheet languages are declarative languages, and hence computations are specified by providing arguments to operations. It is not impossible for a spreadsheet language to support higher-order functions (for example, see [8]), but since higher-order functions are not commonly associated with spreadsheets, for the purposes of this paper we will consider only first-order functions. The restriction to first-order functions makes rich type systems feasible, such as by including use of dependent types. Taking advantage of this opportunity, the unit system presented here includes a similar notion: dependent units.

In spreadsheets, the distinction between "static" and "dynamic" is subtly different than in other languages, because spreadsheets automatically execute after every edit through the "automatic recalculation" feature. Thus, each static edit is immediately followed by dynamic computations. We choose to ignore this advantage to some extent, because by staying only with static (but incremental) devices, the reasoning mechanism can function the same under both eager and lazy evaluation mechanisms, since it will not depend on which cells have been evaluated. Note that the only "input" device for spreadsheets is the entry of constant formulas into cells, and we will consider the evaluation of constant formulas to be a static operation, at least for constant formulas that identify new units. Because "input values" are thus available to the static reasoning system, dependent units are feasible.

The question may arise as to why we have chosen to make the reasoning system static if the intertwining of static-time operations with runtime operations means that complete runtime information is available. The answer is that we would like the reasoning system to work regardless of a spreadsheet's evaluation mechanism. It should work even with lazy evaluation, in which off-screen cells are not computed until they are scrolled on-screen or needed for on-screen calculations. Also, we would like the reasoning system to work even if the user turns off the "automatic recalculation" feature, such as to load a legacy spreadsheet primarily to check it for unit errors.

Still, to be consistent with spreadsheets' default behavior of automatically updating after every edit, a design constraint of the reasoning system is that it must support immediate feedback. Thus, we require the following design constraint to be met:

> **Constraint 1** (*Incremental*): 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.

Recent studies in the realm of one type of problem-solving, namely web searching, indicate that users performed consistently better if they had a basic understanding of the system's selection mechanism [1]. Further, there needs to be a meaningful way to communicate with users about the errors the reasoning system detects. What both of these points suggest is that the reasoning system itself should be in terms of concrete elements explicitly put in the spreadsheet by the user, because the user is familiar with these elements. We have decided to adopt this approach, and we state it as design constraint 2:

> **Constraint 2** (*Directness*): The reasoning mechanisms should directly be in terms of elements with which the user is working, such as labels and operation names.

Most static type inference systems introduce new vocabularies that relate only abstractly and indirectly to the objects and formulas on the screen. Design constraint 3 follows directly from design constraint 2:

> **Constraint 3** (*Not type based*): The reasoning mechanism should require no formal notion of types per se other than what is expressed by units.

Another important difference between other applicative languages and spreadsheet languages is that in spreadsheets, if something goes wrong, such as a (dynamic) type error or a divide-by-zero, computations still continue. This is different from other declarative languages featuring static type checking, which do not allow execution until all type errors are removed. This implies that a reasoning system for spreadsheets must allow reasoning to happen even when a type error (or unit error, in our case) is present somewhere in the spreadsheet.

Further, since end users do not have the training professional programmers have, many of their spreadsheets have characteristics of which computer scientists would not approve. We have a collection of real spreadsheets gathered from

office workers, professors, and a variety of web pages. These spreadsheets show inconsistent labeling, odd layouts, formulas with repeated constants rather then references, cells whose formulas return error values resulting from exceptions [5], and many other oddities. It would not be practical to design an approach purportedly intended for end users that did not work for those kinds of programs, because those are the kinds of programs end users write. We express this point in our fourth design constraint:

> **Constraint 4** (*Practical*): The reasoning mechanism must support the kind of spreadsheets end users really build. In particular, the approach cannot rest upon 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.

3 A Spreadsheet Calculus

In this section we will define a simple model of spreadsheets, the σ-calculus, to have a notation for spreadsheet cells and expressions that can be related to units. The model should be simple enough to facilitate the definition of a unit system and expressive enough to support the end-user requirements discussed in Section 2.

The syntax of the σ-calculus is defined in Figure 2. A spreadsheet (s) consists of a set of named cells. Names, or addresses, (a) are taken from a set N and are distinct from all other values and expressions. Each cell contains an expression (e), which evaluates to a value. A value can be of any suitable type, in particular, values need not be restricted to be numbers or strings. Two distinguished values are the error value ϵ and the blank value $\sqcup$, which are introduced for practical reasons to take into account design constraint 4: spreadsheets often include blank cells and errors. Expressions that are not constants are given by applications of operations to other expressions or by references to other cells. Since we are modeling a first-order language, we do not allow partial applications of operations.

We consider operations (ω) to be indexed by the number of required arguments. Hence, ω^0 ranges over all constants that are different from $\sqcup$ and ϵ. We use v as a synonym for ω^0 whereas $v^{\sqcup}$ ranges over all constants, including $\sqcup$, and v^{ϵ} ranges over all constants, including $\sqcup$ and ϵ. When we use ω^n we implicitly assume $n > 0$ if not explicitly stated otherwise. A spreadsheet is given by a collection of cells whose names are distinct. We can thus regard a spreadsheet as a mapping from names to expressions. We use $dom(s)$ and $rng(s)$ to denote a spreadsheet's domain and range, respectively. More generally, we call a mapping $N \to \alpha$ a *named α-collection*, or just *α-collection* for short. Hence, a spreadsheet is an e-collection.

We deliberately do not require a rectangular structure among the cells since such a restriction is not needed for the investigation performed in this paper and would rule out unnecessarily certain spreadsheet languages, such as Forms/3 [6].

a			Names
e	::=	$\sqcup \mid \epsilon \mid \omega^n(e_1, \ldots, e_n) \mid a$	Expressions ($n \geq 0$)
s	::=	$(a_1, e_1) ; \ldots ; (a_m, e_m)$	Spreadsheets ($m \geq 1$)
		$i \neq j \Rightarrow a_i \neq a_j$	

Fig. 2. The σ-calculus—abstract syntax of spreadsheets.

Spreadsheets (that is, e-collections) like the one shown in Figure 1 (a) are evaluated to v-collections where all expressions have been evaluated to values; see Figure 1 (b). This evaluation is formally defined through a reduction relation given in Appendix A.

In the following we refer to the expression or value contained in a spreadsheet s in the cell a by $s(a)$. Hence, if we name our example spreadsheet *Harvest* and the evaluated sheet *Result*, we have, for example, *Harvest*(D3) = +(B3, C3) and *Result*(D3) = 19. In discussing examples we sometimes identify cells by their content, for example, if we speak of cell Total, we mean the cell named D2.

4 The Nature of 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 $\sqcup$) defines a unit. We can imagine that all the values in a spreadsheet define a unit universe from which the units for cells are drawn. Although all values are units, not all values are generally used as units. For example, in the spreadsheet *Harvest* the text Total is by definition a unit, but it is not used as a unit for cells in the spreadsheet. In the following three subsections we define how basic unit information is provided by headers, how complex units are obtained by unit expressions, and what well-formed units are.

4.1 Headers

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 May and June. In this simple form, headers may seem similar to data types in Haskell or ML, but there are some important differences: first, "constructors" like May might be used as headers for other cells, thus leading to something like "dependent data types". Moreover, a cell might have more than one header, which would correspond to overloaded constructors. For example, the cell B3 has *two* headers, namely Apple and May Another difference is that numbers can be used as constructors. Consider, for example, a variation of the harvest spreadsheet that gives data for different years; see Figure 3. Here, the number 1999 is used as unit for the cells B3, C3, and D3. The formal definition of headers is given in Appendix B.

A header definition represents explicit unit information and is used in the unit checking process. This information has to be provided in some way by the user; it corresponds to type declarations in programming languages. Since we

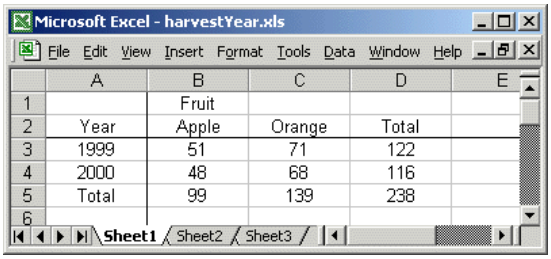


Fig. 3. Yearly production.

cannot expect end users to spend much of their time on declaring units for cells, we have to infer as much unit information as possible automatically or from other user actions that are not directly concerned with units. A header definition can be obtained by exploiting, for instance, the following sources of information:

1. *Predefined unit information.* For instance, the fact that May is a month is known in advance.
2. *Formatting.* For example, if a user devises a specific table format for part of a spreadsheet, unit information can be obtained automatically from the borders of the table.
3. *Spatial & content analysis.* By analyzing a spreadsheet with respect to different kinds of cell contents (for example, text, constant numbers, formulas) and their spatial arrangements, regions can be identified that contain header information (typically the left and top part of tables), footer information (sum formulas at the end of rows and columns), and table data (the constant numbers in between).

In this paper we are not concerned with the process of "header inference"; we assume that the header information is given through the mentioned mechanisms.

4.2 Unit Expressions

We distinguish between simple and complex units. In particular, values like Month or Fruit that do not themselves have a unit are *simple units*. In contrast, *complex units* can be constructed from other units in three different ways to be explained in more detail below.

1. Since units are values, they can themselves have units; hence, we can get chains of units called *dependent units*.
2. 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*.
3. 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 next three subsections we will discuss each of the described unit forms.

Dependent Units Consider the header information for the spreadsheet from Figure 1, shown as a table in Figure 4:

	A	B	C	D
1				
2		B1	B1	
3	A2	A3, B2	A3, C2	
4	A2	A4, B2	A4 ,C2	
5				

Fig. 4. Header information for the *Harvest* spreadsheet.

We observe in the table that B3 has B2 as a header which in turn has B1 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]] (see also Appendix B).

Dependent units express a hierarchy of units that results from the possibility of values being and having units at the same time. The topmost level of this hierarchy is given by the unit **1**, which is not a value and does not have a unit itself. **1** is assigned to all cells for which no more specific unit can be inferred. In a dependent unit, **1** can appear only at the outermost position since it is the unit of all nondependent units. (In other words, unit expressions like $u[\mathbf{1}]$ do not make sense since **1** is not a value and cannot have a unit u.)

The dependencies given by a header definition h define a directed graph whose nodes are cell names and whose edges are given by the set $\{(a, a') \mid a' \in h(a)\}$, that is, edges are directed toward headers. We require this dependency graph to be acyclic. More specifically, we require that all nodes except the roots (sources) have at most one outgoing edge. (This constraint ensures that the unit for each value is given by a simple path.)

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 [15]. In our example, C3 has the unit Fruit[Orange]&Month[May]. It should be clear that the order of units does not matter in an *and* unit. Likewise, the &-unit operator is associative.

An *and* unit of dependent units that have a prefix in common is meaningless and represents an error because subunits define different alternatives for their superunit that exclude each other. For example, it makes no sense for a cell to have the unit Fruit[Apple]&Fruit[Orange] because a number cannot represent numbers of oranges and apples at the same time; such a unit is a contradiction

in itself. The same is true for units like Fruit[Apple[Green]]&Fruit[Apple[Red]] and Fruit[Apple[Green]]&Fruit[Orange]. In contrast, a unit like Month[May]&Fruit[Orange] is reasonable because a number can be classified according to different unit hierarchies.

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 corresponds to the σ-calculus expression $+(\mathsf{B3}, \mathsf{C3})$. 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]. This unit expression can be transformed by commutativity of &, by distributivity of & over |, and prefix factoring for | to Month[May]&Fruit[Apple|Orange]; see Section 5.3. 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.

4.3 Well-Formed Units

The preceding discussion leads to the following definition of unit expressions:

$$u \quad ::= \quad v \;\mid\; u_1[u_2] \;\mid\; u_1 \& u_2 \;\mid\; u_1 | u_2 \;\mid\; \mathbf{1} \;\mid\; \epsilon$$

In addition to the context-free syntax of units, the previously described constraints can be formalized through the concept of *well-formed units* that is defined with respect to a spreadsheet s and a header definition h. 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, and in Figure 3, Year[1999] 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 Apples 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|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.

These rules are formalized in Appendix B.

The described constraints are also a reason for *not* employing a subset model for units: if units were interpreted as sets and dependent units as subsets, we would expect *or* and *and* units to behave like set union and set intersection, respectively. In such a model we certainly require, for example, $\mathsf{Apple} \cap \mathsf{Apple} = \mathsf{Apple}$, however, we have just seen that the corresponding *and* unit is meaningless, which demonstrates that a simple (sub)set/lattice model for units is not adequate.

Not all of a spreadsheet's legal units are actually used as units. For example, in Figure 1, Total is a value and thus also a unit, but it is not used as a unit for another value. Likewise, the well-formed unit Fruit[Apple[8]] is not used in the spreadsheet.

5 A Spreadsheet Unit System

We have defined what spreadsheets and units are; next we have to describe how units are inferred for cells in a spreadsheet.

We need to consider three kinds of judgments: first, we have judgments of the form $(a, e) :: u$ that associate units to cells and that can exploit header information. Second, for the unit inference for operations we also need judgments $e : u$ that give units for expressions regardless of their position (context). Third, for expressing the units of a spreadsheet we need a further kind of judgment $(a_1, e_1)\,;\,\ldots\,;\,(a_m, e_m) ::: (a_1, u_1)\,;\,\ldots\,;\,(a_m, u_m)$, which expresses the fact that the unit information for a spreadsheet is given by a u-collection, also called a *unitsheet*.

We have not yet integrated type checking into unit checking. Recall that by design constraints 2 and 3, we deliberately avoid including a notion of abstract types in the reasoning system for units. We are currently investigating ways of treating types as units, so that type checking and unit checking are communicated in the same way to the end user. However, this approach complicates the structure of dependent units (they are no longer paths but DAGs) and seems to make the unit inference more difficult.

5.1 Unit Inference Rules

The main rules for inferring units for cells are given below. The rule names are given to help reference the formal rules in Appendix C and the examples discussed in Section 5.3.

VAL$_{::}$ All cells that do not have a header, have the unit **1**. For example, cell Total has unit **1**.

DEP$_{::}$ 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 containing 8 whose unit is Fruit[Apple]&Month[May].

REF$_{::}$ 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 $u_a \& u_b$.

APP$_{::}$ 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^n, 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.

A proper treatment of multiplication and division requires the concept of dimensions [17, 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. Therefore, we equip operations like $*$ and / with a rather weak unit transformation: basically, the unit of a product is given by the unit of one factor if all other factors have unit **1**, otherwise the unit is weakened to **1**. In contrast, the unit of a division is always given by the unit of its dividend.[2] If a header definition is present for the cell containing an operation ω, the corresponding unit u is taken into account by creating an *and* unit of u and the computed unit u'. This *and* unit implements a further level of unit consistency checking.

The definition of μ_ω is shown for some operations in Figure 5 (we also abbreviate $u_1, \ldots, u_n$ by $\bar{u}$). u is the cell's header unit; $\bar{u}$ are all the units of the parameters.

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

Fig. 5. Unit transformations.

The unit transformations show how operations are defined for different, but related, units and expose a kind of polymorphism, which we call *unit polymorphism*. Unit polymorphism is similar to parametric polymorphism in the sense that the operations use the same implementation regardless of the units of their arguments. The hierarchy of units also reminds of inclusion polymorphism [7].

[2] Division is not commutative, hence choosing one particular operand is justified. Moreover, the dividend is dominant in specifying the unit, for example, if we had to decide on the unit of "salary per month", we would probably choose "salary".

5.2 Unit Simplification

We can observe that 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. The complete set of equations is given in Appendix C. Some important cases are:

- *Commutativity.* The order of arguments in *or* and *and* units does not matter. For example, the expressions Fruit[Apple]&Month[May] and Month[May]&Fruit[Apple] denote the same unit.
- *Generalization.* A dependent unit $u[u_1|\ldots|u_k]$ whose innermost unit expression is given by an *or* unit that contains *all* the units $u_1, \ldots, u_k$ that have u as a header denotes the same unit as u. For example, the unit expression Fruit[Apple|Orange] expresses the same unit information as just Fruit since the two cells Apple and Orange are the only cells that have Fruit as a header.
- *Factorization.* An *or* unit whose arguments share a common prefix is identical to the unit in which the *or* unit is moved inside. For example, the unit expression Fruit[Apple]|Fruit[Orange] expresses the same unit information as Fruit[Apple|Orange] (which in turn is equal to Fruit).
- *Distributivity* (of *and* over *or*). If one argument of an *and* unit is an *or* unit, then the *and* can be moved into the *or* expression. Likewise, *and* units with a common unit can be moved out of an *or* unit. For example, if a number represents May apples or May oranges (that is, has the unit Month[May]&Fruit[Apple]|Month[May]&Fruit[Orange]), then we can as well say that it represents apples or oranges and at the same time a number for May, which is expressed by the unit Month[May]&(Fruit[Apple]|Fruit[Orange]). Note, however, that *or* units do not distribute over *and*.

5.3 Unit Checking in Action

Next we demonstrate the unit inference rules and the unit simplifications with the example spreadsheet from Figure 1 and the corresponding header information from Figure 4. We have combined both tables and show the spreadsheet with some of its headers represented as arrows in Figure 6.

Since B1 has no header definition, Fruit has unit **1** by rule $\textsc{val}_{::}$; as a judgment this is written $(\mathsf{B1}, \mathsf{Fruit}) :: \mathbf{1}$. Then by rule $\textsc{dep}_{::}$ we know that Orange has the unit $\mathbf{1}[\mathsf{Fruit}]$ (or: $(\mathsf{C2}, \mathsf{Orange}) :: \mathbf{1}[\mathsf{Fruit}]$) because

- the header of C2 is B1,
- the value of the cell B1 is Fruit (that is, *Harvest*(B1) = Fruit), and
- cell Fruit has unit **1**.

Since all units except **1** are of the form $\mathbf{1}[\ldots]$ we omit the leading **1** for brevity in the following. Hence we also say: $(\mathsf{C2}, \mathsf{Orange}) :: \mathsf{Fruit}$. Similarly, we can reason

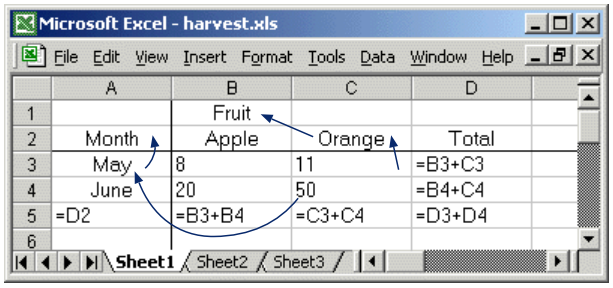


Fig. 6. Some headers for the *Harvest* spreadsheet.

that $(\mathsf{A3}, \mathsf{May}) :: \mathsf{Month}$. Finally, using the last two results and the fact that the header of C3 is defined to be the two cells A3 and C2 we can conclude by rule $\text{DEP}_{::}$ that $(\mathsf{C3}, 11) :: \mathsf{Month}[\mathsf{May}]\&\mathsf{Fruit}[\mathsf{Orange}]$.

Next we infer the unit for the cell D3. First, we have to infer the unit $(\mathsf{B3}, 8) :: \mathsf{Month}[\mathsf{May}]\&\mathsf{Fruit}[\mathsf{Apple}]$, which can be obtained in the same way as the unit for the cell C3. To infer the unit of D3 we apply rule $\text{APP}_{::}$. We already have the units for both arguments; since we have no header information for D3, u is **1** in this case. So we can apply μ_+ and obtain as a result an *or* unit for D3, namely:

$$\mathsf{Month}[\mathsf{May}]\&\mathsf{Fruit}[\mathsf{Apple}]|\mathsf{Month}[\mathsf{May}]\&\mathsf{Fruit}[\mathsf{Orange}]$$

We can now perform rule simplification several times, first exploiting distributivity, which yields the unit:

$$\mathsf{Month}[\mathsf{May}]\&(\mathsf{Fruit}[\mathsf{Apple}]|\mathsf{Fruit}[\mathsf{Orange}]).$$

then we apply factorization, which yields the unit:

$$\mathsf{Month}[\mathsf{May}]\&\mathsf{Fruit}[\mathsf{Apple}|\mathsf{Orange}].$$

Finally, using generalization, we obtain the unit:

$$\mathsf{Month}[\mathsf{May}]\&\mathsf{Fruit}.$$

On the other hand, a cell with an expression $+(\mathsf{B3}, \mathsf{C4})$ would lead to a unit error because we get two different months and two different fruits, which prevents the application of the distributivity rule and thus prevents the unit expression from being reduced to a well-formed unit. Therefore, we cannot infer a unit for such a cell, so the system reports a unit error for that cell.

5.4 Unit Safety

The traditional approach of showing soundness for type systems works for our unit system, too, but it does not yield a very powerful result. (See Appendix C for a formal treatment.) The problem is that soundness does not capture the essential contribution of the unit system, namely preventing unit errors.

The reason is that the operational semantics is defined on values that are not differentiated by units. In other words, computations that are unit incorrect still yield reasonable values since the unit information is ignored by the semantics.

Hence, we need a semantics of values and functions that operate on a finer granularity than types. We can achieve this by tagging values with unit information and defining the semantics in such a way that error values are returned for applications of operations that yield non-well-formed units with their results. The semantics definition for operations must incorporate the unit transformation and the simplification rules (like generalization and factorization). The details will be investigated in a future paper.

6 Conclusions and Future Research

In this paper, we have presented an 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. Significant features of the approach are:

- With units, reasoning about application of operations happens on a more fine-grained level than types.
- The reasoning system includes dependent units.
- The reasoning system, while not supporting parametric polymorphism (for *types*), supports a kind of polymorphism of units.
- The reasoning system is intended for end-user programming of spreadsheets.

In our approach, unit information is given explicitly, which is one reason for the expressiveness of the unit system and for the existence of a simple unit checking procedure. Nevertheless, the often-cited problem of inherent verbosity of explicit type disciplines is not a problem in our approach since the unit information is already present (for example, for documentation) and can be reused. In a sense, our approach can be described as having "implicitly explicit units".

In future work, we will investigate the unit-aware semantics and corresponding unit safety results for the unit system. Moreover, we will investigate the possibilities for header inference. A particular problem is how we can minimize the interaction with the user while trying to get as much unit information as possible. Furthermore, we will try to find a stronger unit treatment of operations requiring dimensions.

References

1. N. Belkin. Helping People Find What They Don't Know. *Communications of the ACM*, 41(8):58–61, 2000.
2. B. Boehm and V. Basili. Gaining Intellectual Control of Software Development. *Computer*, 33(5):27–33, 2000.

3. B. Boehm and V. Basili. Software Defect Reduction Top 10 List. *Computer*, 34(1):135–137, 2001.
4. B. W. Boehm, C. Abts, A. W. Brown, S. Chulani, K. C. Bradford, E. Horowitz, R. Madachy, D. J. Reifer, and B. Steece, editors. *Software Cost Estimation with COCOMO II*. Prentice-Hall International, Upper Saddle River, NJ, 2000.
5. M. M. Burnett, A. Agrawal, and P. van Zee. Exception Handling in the Spreadsheet Paradigm. *IEEE Transactions on Software Engineering*, 26(10):923–942, 2000.
6. M. M. Burnett, J. Atwood, R. Djang, H. Gottfried, J. Reichwein, and S. Yang. Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm. *Journal of Functional Programming*, 11(2):155–206, 2001.
7. L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys*, 17(4):471–522, 1985.
8. W. de Hoon, Rutten L., and M. van Eekelen. Implementing a Functional Spreadsheet in CLEAN. *Journal of Functional Programming*, 5(3):383–414, 1995.
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. 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.
12. 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.
13. R. Panko. Finding Spreadsheet Errors: Most Spreadsheet Models Have Design Flaws that May Lead to Long-Term Miscalculation. *Information Week*, (May 29):100, 1995.
14. R. Panko. What We Know about Spreadsheet Errors. *Journal of End User Computing*, (Spring), 1998.
15. B. C. Pierce. Intersection Types and Bounded Polymorphism. *Mathematical Structures in Computer Science*, 7(2):129–193, 1997.
16. T. Teo and M. Tan. Quantitative and Qualitative Errors in Spreadsheet Development. In *30th Hawaii Int. Conf. on System Sciences*, pages 25–38, 1997.
17. 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.

Appendix

A Operational Semantics of Spreadsheets

The operational semantics of spreadsheets is defined through a reduction relation ($\twoheadrightarrow$) on cell expressions that depends on the definition of the evaluation of operations ($\rightarrow$), which, in particular, has to include a specification of the behavior of operations with respect to the value $\sqcup$; see Figure 7.

The reduction relation is defined relatively to a spreadsheet s, which does not change during evaluation and can thus be treated as a global variable. All cell expressions that cannot be reduced to a value according to the reduction relation

$$
\begin{array}{lll}
+^n(v_1^{\sqcup},\ldots,v_n^{\sqcup}) & \to x_1+\ldots+x_n & \textbf{where } x_i = \textbf{ if } v_i^{\sqcup}=\sqcup \textbf{ then } 0 \textbf{ else } v_i^{\sqcup} \\
\mathsf{count}^n(v_1^{\sqcup},\ldots,v_n^{\sqcup}) & \to x_1+\ldots+x_n & \textbf{where } x_i = \textbf{ if } v_i^{\sqcup}=\sqcup \textbf{ then } 0 \textbf{ else } 1 \\
{}^n(v_1^{\sqcup},\ldots,v_n^{\sqcup}) & \to x_1*\ldots*x_n & \textbf{where } x_i = \textbf{ if } v_i^{\sqcup}=\sqcup \textbf{ then } 1 \textbf{ else } v_i^{\sqcup} \\
\ldots & & \\
\omega^n(\ldots,\epsilon,\ldots) & \to \epsilon &
\end{array}
$$

Fig. 7. Reduction of basic operations.

are defined to reduce to ϵ. This applies, for example, to definitions containing cycles.

The reduction relation $\twoheadrightarrow$ extends naturally to a spreadsheet by application to all of its cells; see Figure 8. A reduced spreadsheet is a v^ϵ-collection, which we also call a *valuesheet*, an example of which was shown in Figure 1 (b).

$$
\begin{array}{lll}
\text{VAL}_{\twoheadrightarrow} \quad \dfrac{}{v^\epsilon \twoheadrightarrow v^\epsilon} & \text{REF}_{\twoheadrightarrow} \quad \dfrac{s(a) \twoheadrightarrow v^\epsilon}{a \twoheadrightarrow v^\epsilon} & \text{REF}^\epsilon_{\twoheadrightarrow} \quad \dfrac{a \notin dom(s)}{a \twoheadrightarrow \epsilon} \\[2ex]
\text{APP}_{\twoheadrightarrow} \quad \dfrac{e_i \twoheadrightarrow v_i^\epsilon \qquad \omega^n(v_1^\epsilon,\ldots,v_n^\epsilon) \to v^\epsilon}{\omega^n(e_1,\ldots,e_n) \twoheadrightarrow v^\epsilon} & \text{APP}^\epsilon_{\twoheadrightarrow} \quad \dfrac{k \neq n}{\omega^n(e_1,\ldots,e_k) \twoheadrightarrow \epsilon} &
\end{array}
$$

$$
\text{SHEET}_{\twoheadrightarrow} \quad \frac{e_1 \twoheadrightarrow v_1^\epsilon \quad \ldots \quad e_m \twoheadrightarrow v_m^\epsilon}{(a_1,e_1)\,;\ldots;\,(a_m,e_m) \twoheadrightarrow (a_1,v_1^\epsilon)\,;\ldots;\,(a_m,v_m^\epsilon)}
$$

Fig. 8. Operational semantics of spreadsheets.

A spreadsheet is said to be ϵ-*free* if its reduction does not contain an error value, that is, $s = (a_1,e_1)\,;\ldots;\,(a_m,e_m)$ is ϵ-free $:\Longleftrightarrow s \twoheadrightarrow (a_1,v_1^{\sqcup})\,;\ldots;\,(a_m,v_m^{\sqcup})$.

B Definition of Headers and Units

We formalize the notion of header through a $\{a\}$-collection, called *header definition*: $h(a) = \{a_1,\ldots,a_k\}$ means that cell a has the cells $a_1,\ldots,a_k$ as headers.[3]

Unit expressions are defined by the grammar given in Section 4.3. The syntax allows unit expressions for dependent units to be arbitrary binary leaf trees, but we consider all trees that have the same order of leaves to be equal; see also Figure 11. We select the right-spine tree, which corresponds to the unit expression $u_1[u_2[\ldots u_{n-1}[u_n]\ldots]]$, as the canonical representative for the class of all equivalent dependent units.

Well-formed units are expressed by judgments $\vdash u$ defined in Figure 9. For notational brevity we omit the spreadsheet s and the header definition h and consider them as global constants since they are changed within the rules.

[3] We require $k > 0$ since $h(a) = \emptyset$ would be interpreted in the same way as the case $a \notin dom(h)$.

$$\text{ONE}_{\vdash}\ \frac{}{\vdash \mathbf{1}} \qquad \text{VAL}_{\vdash}\ \frac{a \notin dom(h) \qquad s(a) = v}{\vdash \mathbf{1}[v]}$$

$$\text{DEP}_{\vdash}\ \frac{a' \in h(a) \qquad s(a) = v \qquad \vdash u \qquad \vdash u[s(a')]}{\vdash u[s(a')[v]]}$$

$$\text{AND}_{\vdash}\ \frac{\vdash \mathbf{1}[u_1[\ldots u_{n-1}[u_n]\ldots]] \qquad \vdash \mathbf{1}[u'_1[\ldots u'_{m-1}[u'_m]\ldots]] \qquad u_i \neq_u u'_i}{\vdash u_1[\ldots u_{n-1}[u_n]\ldots] \& u'_1[\ldots u'_{m-1}[u'_m]\ldots]}$$

$$\text{OR}_{\vdash}\ \frac{\vdash \mathbf{1}[u_1[\ldots u_n[v_1]\ldots]] \ \ldots \ \vdash \mathbf{1}[u_1[\ldots u_n[v_k]\ldots]] \quad n > 1 \quad k > 1 \quad v_i \text{ distinct}}{\vdash \mathbf{1}[u_1[\ldots u_n[v_1|\ldots|v_k]\ldots]]}$$

Fig. 9. Well-formed units.

C Unit Inference

We can give the set of inference rules for determining units for (all cells of) a spreadsheet, formalizing the examples given in the previous section. See Figure 10. As in the definition for well-formed units, we regard s and h as global constants.

$$\text{VAL}_{:}\ \frac{}{v^{\sqcup} : \mathbf{1}} \qquad \text{REF}_{:}\ \frac{(a, s(a)) :: u \qquad \vdash u}{a : u} \qquad \text{EQ}_{:}\ \frac{e : u \qquad u =_u u'}{e : u'}$$

$$\text{APP}_{:}\ \frac{e_i : u_i \qquad \vdash u_i}{\omega^n(e_1, \ldots, e_n) : \mu_\omega(\mathbf{1}, u_1, \ldots, u_n)}$$

$$\text{VAL}_{::}\ \frac{a \notin dom(h)}{(a, v^{\sqcup}) :: \mathbf{1}} \qquad \text{REF}_{::}\ \frac{(a, s(a)) :: u \quad (a', s(a')) :: u' \quad \vdash u \quad \vdash u'}{(a, a') :: u \& u'}$$

$$\text{APP}_{::}\ \frac{e_i : u_i \qquad \vdash u_i \qquad (a, \sqcup) :: u \qquad \vdash u}{(a, \omega^n(e_1, \ldots, e_n)) :: \mu_\omega(u, u_1, \ldots, u_n)} \qquad \text{EQ}_{::}\ \frac{(a, e) :: u \quad u =_u u'}{(a, e) :: u'}$$

$$\text{DEP}_{::}\ \frac{h(a) = \{a_1, \ldots, a_k\} \qquad s(a_i) = v_i^{\sqcup} \qquad (a_i, v_i^{\sqcup}) :: u_i \qquad \vdash u_i}{(a, v^{\sqcup}) :: u_1[v_1^{\sqcup}] \& \ldots \& u_k[v_k^{\sqcup}]}$$

$$\text{SHEET}_{:::}\ \frac{i \in J_m \qquad (a_i, e_i) :: u_i \qquad \vdash u_i \qquad j \in \bar{J}_m \qquad u_j = \epsilon}{(a_1, e_1)\,;\ \ldots\ ;\,(a_m, e_m) ::: (a_1, u_1)\,;\ \ldots\ ;\,(a_m, u_m)}$$

Fig. 10. A unit system for the σ-calculus.

The rules $\text{EQ}_{:}$ and $\text{EQ}_{::}$ exploit the equations shown in Figure 11, which define equality of units. (Note that & distributes over |, but not vice versa, and that although $\mathbf{1}$ is the unit for &, it leads to a non-valid unit when combined with |.)

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. Moreover, this condition must hold for all definitions (copies) of u (that is, for all cells containing u):

$u_1 \& u_2$	$=_u$	$u_2 \& u_1$	(commutativity)
$u_1 \mid u_2$	$=_u$	$u_2 \mid u_1$	
$(u_1 \& u_2) \& u_3$	$=_u$	$u_1 \& (u_2 \& u_3)$	(associativity)
$(u_1 \mid u_2) \mid u_3$	$=_u$	$u_1 \mid (u_2 \mid u_3)$	
$u \& (u_1 \mid u_2)$	$=_u$	$(u \& u_1) \mid (u \& u_2)$	(distributivity)
$u \& u$	$=_u$	u	(idempotency)
$u \mid u$	$=_u$	u	
$\mathbf{1} \& u$	$=_u$	u	(unit)
$u[u_1] \mid u[u_2]$	$=_u$	$u[u_1 \mid u_2]$	(factorization)
$u[u_1 \mid \ldots \mid u_k]$	$=_u$	$u \qquad \Leftarrow (*)$	(generalization)
$(u_1[u_2])[u_3]$	$=_u$	$u_1[u_2[u_3]]$	(linearization)

Fig. 11. Unit equality.

$$\forall a \in s^{-1}(u) : h^{-1}(\{a\}) = \{a_1, \ldots, a_k\} \ \wedge \ s(a_1) = u_1 \ \wedge \ \ldots \ \wedge \ s(a_k) = u_k$$

In the rule $\text{SHEET}_{:::}$ we use the notation J_m for a subset of $\{1, \ldots, m\}$ and $\bar{J}_m$ to denote $\{1, \ldots, m\} - J_m$. The rule $\text{SHEET}_{:::}$ can be used to derive different u-collections for one spreadsheet. We can define an ordering on the u-collections based on the number of ϵ-values contained in them, so that we can select the u-collection containing the least number of ϵ-values as *the* u-collection for a spreadsheet (which we could call the spreadsheet's *principal u-collection.*)

The fact that the principal u-collection is always uniquely defined follows from the structure of the unit system: the unit for each cell can be computed in a syntax-directed way without having to make any choices, so that it is not possible, for example, to remove a unit error in one cell by "allowing" an error in another cell.

For completeness, we mention as a straightforward result that the defined unit system is sound. First, we have the following lemma for individual cells:

Lemma 1 (Unit Soundness for Cells). $(a, e) :: u \wedge \vdash u \Longrightarrow e \twoheadrightarrow v^{\sqcup}$

Since reduction and the units for a spreadsheet are directly derived from the corresponding relations on cells, the unit soundness for spreadsheets follows directly from Lemma 1:

Corollary 1 (Unit Soundness for Spreadsheets).
$(a_1, e_1) \,;\, \ldots \,;\, (a_m, e_m) ::: (a_1, u_1) \,;\, \ldots \,;\, (a_m, u_m) \wedge \vdash u_i \Longrightarrow$
$(a_1, e_1) \,;\, \ldots \,;\, (a_m, e_m) \twoheadrightarrow (a_1, v_1^{\sqcup}) \,;\, \ldots \,;\, (a_m, v_m^{\sqcup})$

In other words, a spreadsheet that does not contain unit errors is guaranteed to be ϵ-free.