Type Inference for Spreadsheets *

Robin Abraham   Martin Erwig

School of EECS
Oregon State University
[abraharo|erwig]@eecs.oregonstate.edu

Abstract

Spreadsheets are the most popular programming systems in use today. Since spreadsheets are visual, first-order functional languages, research into the foundations of spreadsheets is therefore a highly relevant topic for the principles and, in particular, the practice, of declarative programming.

Since the error rate in spreadsheets is very high and since those errors have significant impact, methods and tools that can help detect and remove errors from spreadsheets are very much needed. Type systems have traditionally played a strong role in detecting errors in programming languages, and it is therefore reasonable to ask whether type systems could not be helpful in improving the current situation of spreadsheet programming.

In this paper we introduce a type system and a type inference algorithm for spreadsheets and demonstrate how this algorithm and the underlying typing concept can identify programming errors in spreadsheets. In addition, we also demonstrate how the type inference algorithm can be employed to infer models, or specifications, for spreadsheets, which can be used to prevent future errors in spreadsheets.

Categories and Subject Descriptors F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Type Structure; H.4.1 [*Information Systems Applications*]: Office Automation—Spreadsheets

General Terms Languages, Design

Keywords Type Inference, Templates, End-User Software Engineering

1. Introduction

Functional programming is by far the most popular programming paradigm, one could argue, considering that spreadsheets, which are (first-order) functional programs [32], are the most widely used programming systems in the world. It is estimated that each year tens of millions of professionals and managers create hundreds of millions of spreadsheets [31]. The popularity of spreadsheet does, however, not tell much about their reliability. In fact, many studies have shown that existing spreadsheets contain many errors [8, 25, 31]. Some estimates even even suggest that 90% or more of real-world spreadsheets contain errors [35]. Moreover, many of these errors have significant impact, causing business losses in the billions [21, 17].

This observation seems to be in stark contrast to some of the claims of functional programming advocates, for example, that functional programs are more reliable than, for example, imperative programs, and contain fewer errors. However, a closer look reveals that the increased reliability of functional programs is achieved, at least to some degree, through a cleaner language design, offering powerful abstractions, such as higher-order functions, and through sophisticated type systems that help to detect program errors early.

Unfortunately, spreadsheets lack both, which might be one of the reasons for the high error incidence. Therefore, a strategy to significantly improve the reliability of spreadsheets could be to add (a) higher-order programming constructs and/or (b) a type system to spreadsheets. The first approach is questionable because no convincing proposals for visual representations of higher-order functions, suitable for end users, have been made to date, and simply adding a textual layer on top of spreadsheets impacts their highly attractive visual interface with its immediate feedback [22, 26, 28, 24]. The second approach seems to be more promising. In fact, there have been several proposals for type systems for spreadsheets that are based on labels that users place in spreadsheets.

In [16] we have introduced a type system for spreadsheets that is based on the idea to associate cells with so-called *units*, which are given by labels users have used in the spreadsheet. Other research has been built on this idea [6, 11], demonstrating that unit reasoning can give valuable information about a spreadsheet and possible errors in it. We have implemented these ideas together with algorithms for automatic header inference (the process of identifying relationships between cells and labels) in a system called UCheck [1]. The rationale for the unit approach is to be able to communicate errors to end users in a form that they understand, that is, in terms of units/labels that they themselves are using in their spreadsheets and not in terms of abstract type system jargon [3].

In contrast, type systems in the traditional sense have not been considered before for spreadsheets. The reason might be that due to the lack of polymorphism[1] and higher-order

* This work is partially supported by the National Science Foundation under the grant ITR/AP-0121542 and by the EUSES consortium (`EUSESconsortium.org`).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPDP'06 July 10–12, 2006, Venice, Italy.

Copyright © 2006 ACM 1-59593-388-3/06/0007...$5.00.

[1] Although, as we will see, references do have polymorphic types.

types, those type systems seem to be ridiculously simple and not worth being investigated at all.

However, as we will demonstrate in this paper, non-trivial typing concepts can be identified for formulas, cells, and spreadsheets, and type systems and type inference can contribute significantly to improving the reliability of spreadsheets. We will consider, in particular, the following two applications.

- *Identifying errors in spreadsheets.* This is the classical motivation for type systems. In the case of spreadsheets, a surprising variety of errors can be detected using type inference.
- *Providing more accurate model inference.* Model inference is concerned with identifying a general model of which a particular spreadsheet is an instance. Knowledge of a spreadsheet's model can help to identify errors in the current spreadsheet, but maybe even more important, it can be exploited to *prevent* a large class of errors in future versions of the spreadsheet [14, 5, 15].

The essential ideas of our approach to defining a type system for spreadsheets can be summarized as follows. First, we define the type of formulas as *function types*, in which the argument types are given by the types that are expected for the formula's external references. The result type is given by the result type of the outermost operation. Second, we define a *cell type* as a pair consisting of the cell formula's result type and a mapping from its referenced cells to possible *type conflicts*, which are given by pairs of expected and actually found types. Third, extending cell types to spreadsheets leads to spreadsheet types that give a detailed overview of the type correctness in different parts of a spreadsheet. Finally, we define *template types* as a kind of "encoding" of spreadsheet types that can summarize large spreadsheet types succinctly.

In addition to the type system, we also present a type inference algorithm to compute template types. This algorithm is a generalization of an algorithm that we have developed in a different context to infer spreadsheet models [4]. The algorithm is parameterized by an equivalence relationship on cells. This approach allows us to instantiate different template inference algorithms by choosing different cell equivalences, which enables the comparison of different kinds of templates to represent spreadsheet models. In particular, using the concept of *cell type equivalence* leads to the inference of most general template types for spreadsheets. An obvious use of these template types is to report type errors in the spreadsheet. In addition, template types can be used in conjunction with a syntactic notion of spreadsheet templates [4] to support the inference of spreadsheet models. A more accurate description of spreadsheet models based on type equivalence leads, in particular, to a better understanding of spreadsheets to prevent errors (by enforcing models through techniques and tools described in [14, 5, 15]).

The rest of this paper is structured as follows. In Section 2 we will discuss related work. In Section 3 we will illustrate several kinds of errors that are common in spreadsheets, and we will briefly review a previous approach to prevent errors through a template-driven program-generation approach. This work provides the motivation for the second contribution of the type inference algorithm, namely the inference of more accurate templates. In Section 4 we will introduce a simple model of spreadsheets as programs and give syntax and semantics. The spreadsheet type system is then introduced in a series of steps in Section 5, followed by a description of the type inference algorithm in Section 6. In fact, the type inference algorithm is presented as an instance of a more general pattern inference algorithm that can be used for different purposes. We will analyze the possible impact of the type system and the type inference algorithm in Section 7. In particular, we will revisit the examples given in Section 3 and illustrate how the proposed system can help with the described errors. Conclusions given in Section 8 complete this paper.

2. Related Work

Many researchers have looked at errors in programs developed by novice programmers [41, 40, 12]. Even though the benefits of type systems are widely accepted, not many studies have been carried out to test or compare the usability of type systems in general-purpose programming languages. Empirical studies have demonstrated the defect-detection capabilities of static type checking [20, 33]. Even though spreadsheets are among the most widely used programming systems [39], spreadsheets systems like Microsoft Excel do not carry out type checking on spreadsheet formulas.

In previous work, we have developed a system called UCheck [1] that uses the labels within the spreadsheet to carry out so called *unit checking* [16] of the spreadsheet formulas. In the first step of this two-step process of automatic consistency checking, the system infers the labels automatically and assigns units to the input cells on the basis of the inferred labels. UCheck then checks the formulas to ensure that formulas that violate the rules for valid combination of units (as defined in [16]) are flagged as potential sites of faults. The perceived advantage of this approach was that it would be easier for users to understand the errors in the spreadsheet if the labels they themselves had entered were used while reporting the errors. Approaches that use explicit header annotations entered by the user have been presented in [6, 11]. Checking of spreadsheet formulas based on the actual physical or monetary units has been presented in [7]. Evaluations carried out using these systems (as reported in the papers) have shown that these techniques are effective at detecting faults within spreadsheets.

Spreadsheet errors could be the result of any one of the following.

1. Poor understanding of the problem domain.
2. Errors from poor implementation of a solution.
3. Combinations of varying degrees of the two preceding factors.

To help users overcome these problems, we have developed a system that uses spreadsheet templates created using the Visual Template Specification Language (ViTSL) [5] to generate spreadsheets free from reference, range, or type errors [14, 15]. The advantage of this approach is that the templates could be created and verified by domain experts and used by less experienced users to generate spreadsheets that always conform to the template. We have also looked at more expressive specification languages for spreadsheets [13] and have developed a system that allows users to extract ViTSL templates from their legacy spreadsheets [4].

Researchers have also proposed guidelines for designing better spreadsheets so development and maintenance tasks can be carried out with lower risk of introducing errors [36, 42, 23, 34]. However, such techniques are difficult to enforce and involve the cost of training the user. It has also been shown that code inspection and auditing of spreadsheets

[29, 38, 27] fail to detect all the faults. As a matter of fact, it has been shown that group code inspection detects up to 83% of the faults versus an error detection rate of 63% for individual code inspection. Since the users do not get any feedback about the correctness of their spreadsheets after carrying out code inspection or auditing, they come away overconfident that their spreadsheets are error free [30].

The "What You See Is What You Test" (WYSIWYT) approach presented in [37] helps users test spreadsheets. The system uses data-flow adequacy and coverage criteria to give the user feedback on how well tested the spreadsheet is. The "Help Me Test" (HMT) system [19] automatically generates test cases for the spreadsheet the user is working on. The Forms/3 system, of which WYSIWYT and HMT are components, also allows users to define assertions on the expected values within cells [9]. The system propagates these assertions forward and computes the assertions for the output cells. The system warns the user if system-generated and user-specified assertions do not agree or if any assertion is violated by values in cells.

To minimize the number of errors that could be introduced during formula edits, we have developed a spreadsheet debugger, called GoalDebug, that allows users to mark cells with incorrect outputs and specify the expected output [2]. The GoalDebug system then generates a list of change suggestions, any one of which when applied would result in the expected output being computed in the marked cell. We use a set of heuristics to rank the generated change suggestions before presenting them to the user.

3. Example

An example spreadsheet, which is used to keep track of student scores in a course, is shown in Figure 1. For illustration purposes, this spreadsheet has been seeded with many errors. However, it is difficult to determine the faults in the spreadsheet just by looking at the values. This view, which displays the values, is the default in Excel and other spreadsheet systems, and it is difficult to identify cells with errors just by looking at the spreadsheet. The task is complicated even further in the case of large spreadsheets, especially if the person inspecting the spreadsheet for errors is not very clear about the specifications for the spreadsheet.

Microsoft Excel - Grades.xls

	A	B	C	D	E	F	G	H	I	J
1		Assg	1	Assg	2	Assg	3			
2	Name	20		30		15		Average	Grade	Above/Below
3	Ben	19	95%	19	95%	13	87%	95%	P	Above
4	Lisa	17	85%	28	93%	13	87%	89%	P	Above
5	Sue	16	80%	11	7%	14	93%	60%	P	Above
6	Ken	17	85%	12	0%	14	108%	64%	F	Above
7								77.1%		

Figure 1. Grade spreadsheet containing errors.

3.1 Spreadsheet Errors

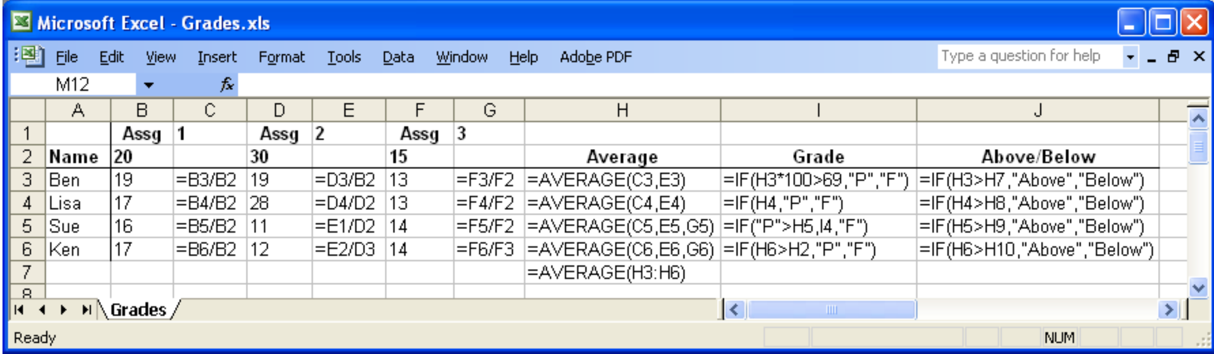
The grade spreadsheet from Figure 1 is shown with the formula view enabled in Figure 2. In the formula view, for this simple and small spreadsheet the errors are relatively easy to spot. We discuss them briefly since they reflect errors that are often present in real-world spreadsheets.

The percentage score for each student for an assignment is to be computed by dividing the student's score by the total points on the assignment. The total points on the assignments one, two, and three are the values in cells B2, D2, and F2, respectively. The overall average scores for Ben and Lisa are incorrect because the formulas in H2 and H3 omit the percentage scores for Assignment 3. Such omission errors can easily occur since the user has to manually update the formulas for the overall averages every time the scores for a new assignment are added to the spreadsheet. The formulas in E3, E5, E6, and G6 have incorrect references in them.

The formulas in column I assign a final grade to each student for the course. Students with an overall average score over 69% pass the course, that is, they get a "P". Otherwise they fail the course, that is, they get an "F". Some of the errors seeded in column I might seem contrived, but it is actually very easy to introduce these kinds of errors into a spreadsheet by unintended cell editing actions. The formula IF(H3*100>69,"P","F"), in I3, is the only correct one in the column. Ben gets a "P" on the course since his overall average score is 95%, which is well above the 69% required to get a "P". The formula in I4 simply has a reference to H4 instead of a condition in the IF statement. While the output "P" is correct in this particular case since Lisa's overall average score in H4 is 91%, the formula would also assign Lisa a "P" grade for all non-zero values, including negative numbers, in H4. The condition in the IF statement in I5 checks if the string "P" is greater than the number computed in H5. Since strings are always greater than numbers in Excel, the condition evaluates to true and Sue is assigned the same grade as Lisa because of the reference to I4 in the THEN branch of the formula. So Sue gets a "P" despite scoring only 60% on the course. The condition in the IF statement in I6 checks if the output number in H6 is greater than the string "Average" in H2. As in the previous case, since strings in Excel are always greater than numbers, the condition evaluates to false, and Ken gets an "F" on the course.

The formulas in J are meant to compare each student's overall average score against the overall average score of the entire class (this score is computed by the formula in H7). The expected output is "Above" if the student's score is above the class average and "Below" otherwise. Only the formula in J3 meets this requirement, the formulas in J4, J5, and J6, compare the student's scores with the empty cells H8, H9, and H10 respectively. All these mistakes could have been introduced easily through the auto-updating of cell formulas during copy-paste or formula-drag. All these comparisons return true and the formulas result in "Above". Obviously, not all students can be have scores that are above the class average.

Quite a few of the errors discussed above would be detected by a rudimentary type checker. For example, a type checker would report the cases in which the condition in the IF statement is not evaluating to a boolean value. Moreover, a type checker would also detect the cases in which dissimilar types are being compared in a condition. For example, a type checker would report an error in the condition comparing a string and a number in the formula in I5, and a number and a blank cell (which has the type Undef in our system) in the formula in J4. These are just some simple cases in which typing would be helpful. We look at this aspect in greater detail in Section 7.



	A	B	C	D	E	F	G	H	I	J
1		Assg	1	Assg	2	Assg	3			
2	Name	20		30		15		Average	Grade	Above/Below
3	Ben	19	=B3/B2	19	=D3/B2	13	=F3/F2	=AVERAGE(C3,E3)	=IF(H3*100>69,"P","F")	=IF(H3>H7,"Above","Below")
4	Lisa	17	=B4/B2	28	=D4/D2	13	=F4/F2	=AVERAGE(C4,E4)	=IF(H4,"P","F")	=IF(H4>H8,"Above","Below")
5	Sue	16	=B5/B2	11	=E1/D2	14	=F5/F2	=AVERAGE(C5,E5,G5)	=IF("P">H5,I4,"F")	=IF(H5>H9,"Above","Below")
6	Ken	17	=B6/B2	12	=E2/D3	14	=F6/F3	=AVERAGE(C6,E6,G6)	=IF(H6>H2,"P","F")	=IF(H6>H10,"Above","Below")
7								=AVERAGE(H3:H6)		

Figure 2. Grade spreadsheet in formula view.

3.2 ViTSL/Gencel Approach to Safe Spreadsheets

Two aspects of the work reported in this paper become clear from the example of the grade spreadsheet. First, due to the absence of systematic error-checking tools in Microsoft Excel, errors are insidious and difficult to detect and correct. Second, especially in the case of large spreadsheets, it becomes more and more difficult to understand the underlying model (or specification) of a spreadsheet. These problems could potentially severely limit a user's ability to audit or even make modifications to a spreadsheet.

In the ViTSL/Gencel approach described in [14], the user would work with a ViTSL template [5] designed by a domain expert. In the case of a grade sheet, a teacher with a working knowledge of spreadsheets could be considered a domain expert. The ViTSL template for the grade sheet is shown in Figure 3.

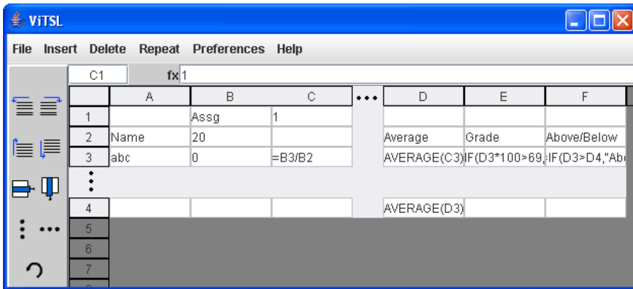


Figure 3. ViTSL template for the grade sheet.

The user can load the template into the Gencel system [14], which generates the first instance of the spreadsheet shown in Figure 4.

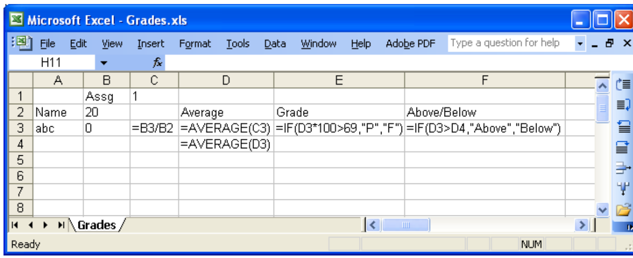


Figure 4. Gencel instance of the grade template.

The user can then use the customized insert/delete row/column operations provided by Gencel to build the spreadsheet. The advantage of this approach is that the system performs all the formula updates automatically so that any instance of the grade spreadsheet generated using Gencel would conform to the specifications expressed as the ViTSL template. Given an error-free ViTSL template, the user only needs to ensure that the values in the input cells are correct, and the system guarantees [15] protection from reference, range, and type errors in all generated instances of the grade spreadsheet.

To enable spreadsheet users to extract templates from their existing spreadsheets, we have developed the Parcel system described in [4], which uses *cp-similarity*. Two cells are cp-similar if their formulas could have resulted from a copy/paste action from one of the cells to the other. The template that is automatically inferred from an error-free version of the grade spreadsheet is shown in Figure 5.

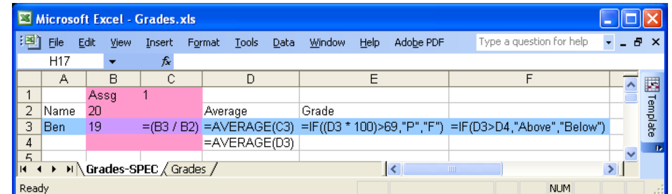


Figure 5. Automatically inferred template for the grade sheet.

The cells that are shaded light blue (A3, D3:F3) can repeat vertically, those shaded pink (B1:C2, B4:C4) can repeat horizontally, and those are shaded purple (B3:C3) can repeat both vertically and horizontally.

4. A Formal Model of Spreadsheet Programs

A spreadsheet is given by a collection of formulas and values embedded into a spatial structure. In most cases this spatial structure is a rectangular grid, whose elements can be addressed by pairs of integers. We can define the set of addresses as $A = \mathbb{N} \times \mathbb{N}$ and use a to range over A. Not only are formulas embedded into the spatial structure, they can also refer to other formulas through cell addresses. In general, we can distinguish between relative and absolute addresses. While absolute addresses can be represented by

elements $a \in A$, relative references are given by offsets $\delta \in \Delta$ where $\Delta = \mathbb{Z} \times \mathbb{Z}$. In most real-world spreadsheet systems combinations of the two addressing modes are also possible. For simplicity we consider in this paper only formulas that contain relative references. Formulas are defined by the following grammar.

$$f \in F \quad ::= \quad v \mid \delta \mid \omega(f, \ldots, f) \qquad \textit{formulas}$$

Values ($v \in V$) include numbers, booleans, and strings, whereas operations (ω) include binary operations, aggregations, etc.

The given definition of formulas makes the concept of cp-similar cells simple: Two formulas are cp-similar if and only if they are equal.

A spreadsheet can be regarded as a partial function that maps cell addresses to formulas.

$$S : A \to F \qquad \textit{spreadsheets}$$

An element $(a, f) \in S$ is called a *cell*. We use the notation $C_S(a) = (a, S(a))$ to yield the cell (that is, address together with the formula) stored at address a in the spreadsheet S. Two cells are cp-equivalent if the stored formulas are, that is, any two cells (a, f) and (a', f) are cp-equivalent.

The evaluation of a cell (a, f) in the context of a spreadsheet is denoted by $[\![(a, f)]\!]_S$ and is defined as follows.

$$\begin{array}{ll} [\![(a, v)]\!]_S & = v \\ [\![(a, \delta)]\!]_S & = [\![C_S(a + \delta)]\!]_S \\ [\![(a, \omega(f_1, \ldots, f_n))]\!]_S & = [\![\omega]\!]([\![(a, f_1)]\!]_S, \ldots, [\![(a, f_1)]\!]_S) \end{array}$$

In the above definition $[\![\omega]\!]$ refers to the function denoted by the predefined operation ω. The semantics of cells containing formulas with circular references is undefined. Finally, the semantics of a spreadsheet S is simply given by the semantics of its cells, that is, $[\![S]\!]_S = \{(a, [\![(a, f)]\!]_S) \mid (a, f) \in S\}$.

5. A Type System for Spreadsheets

In this section we introduce typing concepts for the different elements of a spreadsheet. In particular, we will consider a particular form of typing judgments for formulas and cells that allows a fine-grained typing of spreadsheets by recording individual type violations on the cell level.

The type system is layered into multiple levels. The following table provides an overview and serves as a roadmap through this section.

	Objects	Types
1	values & operations (v, ω)	base types (α, β)
2	formulas (f)	formula types (ϕ)
3	cells (c)	cell types (γ)
4	spreadsheets (S)	spreadsheet types (σ)
5	spreadsheets	template types (τ)

The table indicates that we will introduce two kinds of types for spreadsheets: spreadsheet types and template types.

5.1 Typing of Values and Operations

The first level of the spreadsheet type system assigns base types to values and function types to operations. Therefore, we use the following definition of basic types (β), which include types for constants (α).

$$\begin{array}{lll} \alpha \in T & ::= \quad \mathsf{Num} \mid \mathsf{String} \mid \mathsf{Bool} \mid \mathsf{Undef} & \textit{constant types} \\ \beta & ::= \quad \alpha \mid \alpha \times \ldots \times \alpha \to \alpha & \textit{base types} \end{array}$$

By using α in the type for operations, we effectively restrict the model to first-order operations. Although this restriction is not essential, it reflects the reality of spreadsheets.

We assume that the judgments $v : \alpha$ and $\omega : \beta$ are given for all predefined values and operations.

5.2 Typing of Formulas

We could define typing rules for formulas that simply yield the return type of a formula's outermost operation. In this case, references in formulas have to be type checked by determining the types of the formulas or values that are contained in the referenced cells.

Alternatively, we can also initially consider the type of a formula separately from its embedding into a spreadsheet, which allows us to consider type error situations in more detail. In this case, the references of a formula are paired with the corresponding argument types of the operations which contain them. This information about expected types of referenced cells is called a *type expectation* and is represented by a mapping from addresses to constant types.

$$\Gamma : \Delta \to \alpha \qquad \textit{type expectation}$$

The type of a formula is given by its result type and a type expectation Γ.

$$\phi \quad ::= \quad \Gamma \Rightarrow \alpha \qquad \textit{formula types}$$

The double arrow in the syntax for formula types indicates that the result type is dependent on the type expectation. We need the following operation to combine type expectations.

$$\Gamma \oplus \Gamma' \;=\; \begin{cases} \Gamma \cup \Gamma' & \text{if } \Gamma(\delta) = \alpha \wedge \Gamma'(\delta) = \alpha' \Longrightarrow \alpha = \alpha' \\ \bot & \text{otherwise} \end{cases}$$

Now we can define in Figure 6 the judgment $f|\alpha \succ \Gamma$ that produces a type expectation for the references contained in formula f that is expected to have result type α.

$$\textsc{Val}_\succ \; \frac{v : \alpha}{v|\alpha \succ \varnothing} \qquad \textsc{Ref}_\succ \; \frac{}{\delta|\alpha \succ \{(\delta, \alpha)\}}$$

$$\textsc{Fml}_\succ \; \frac{\omega : \alpha_1 \times \ldots \times \alpha_n \to \alpha \quad f_i|\alpha_i \succ \Gamma_i \quad \Gamma_1 \oplus \ldots \oplus \Gamma_n \neq \bot}{\omega(f_1, \ldots, f_n)|\alpha \succ \Gamma_1 \oplus \ldots \oplus \Gamma_n}$$

Figure 6. Deriving type expectations.

Note that the rules in Figure 6 ensure that formulas are well typed in their use of values and operations.

Next we can define the type of formulas through the judgment $f : \phi$. Note that Γ records the types required for the references contained in f and α is f's result type. The rules are shown in Figure 7. The overloading of the ":" symbol in the typing notation is not problematic since the difference can always be told from the type of participating type arguments (that is, α and β for predefined types and ϕ for formula types).

The typing rule $\textsc{Ref}_:$ indicates that references are actually polymorphic.

We define that a formula f is *type correct* if a derivation exists for the judgment $f : \phi$. Two formulas f and f' are *type equivalent* if they are well typed and have the same type, that is:

$$\exists \phi : f \leftrightarrow_\phi f' \Longleftrightarrow f : \phi \wedge f' : \phi$$

Since cp-similarity means equality of formulas, it follows directly that two cp-equivalent formulas are also type equiva-

$$\text{VAL: } \frac{v : \alpha}{v : \varnothing \Rightarrow \alpha} \qquad \text{REF: } \frac{}{\delta : \{(\delta, \alpha)\} \Rightarrow \alpha}$$

$$\text{FML: } \frac{\omega : \alpha_1 \times \ldots \times \alpha_n \to \alpha \qquad \omega(f_1, \ldots, f_n)|\alpha \succ \Gamma}{\omega(f_1, \ldots, f_n) : \Gamma \Rightarrow \alpha}$$

Figure 7. Typing rules for formulas.

lent. In other words, cp-equivalence of formulas is a strictly stronger criterion than type equivalence of formulas.

LEMMA 1. $f = f' \Longrightarrow \exists \phi : f \leftrightarrow_\phi f'$

Note that cp-similarity is a sufficient, but not a necessary condition for type equivalence of formulas. For example, 3 and 4 are type equivalent, but not cp-equivalent.

This result seems to indicate that type equivalence cannot really contribute anything new (beyond what is already known from cp-similarity) to the semantic analysis of formula relationships. However, the situation will be quite different when we consider in the next section the types of formulas that are embedded in spreadsheets.

5.3 Typing of Cells

When we consider a cell $c = (a, f)$ that contains a type-correct formula of type $\Gamma \Rightarrow \alpha$, then c is type correct in the context of S only if the references in f refer to cells that have the types as required by Γ, that is,

$$\Gamma(\delta) = \alpha \Longrightarrow S(a + \delta) : \alpha$$

Correspondingly, if this condition is violated, the cell is *not* type correct and contains one or more type errors at the offending referenced cells. Now we could define the type of a spreadsheet to be a mapping from addresses to base types, and we could simply define a rule system that succeeds only if all cells are type correct. However, this approach seems to be too rigid, because in cases where most cells except a few are type correct, it would be interesting to know where those type violations occur. Thus the goal should be to define spreadsheet types and a corresponding rule system in a way that they can provide feedback to that effect.

Therefore we define a spreadsheet type more generally to be a mapping from addresses to types and type mismatches, where a *type mismatch* is simply given by a pair of different constant types (α, α') that expresses that a cell contains a value or formula whose result is of type α but is used in some other formula that expects a value of a different type α'. A type mismatch is always associated with two addresses a and a' that contain formulas which "produce" α and "consume" α while expecting α', respectively. Recognizing that a pair (a, α) represents a "type cell" (as opposed to a value or formula cell), we call a pair of type cells $((a, \alpha), (a', \alpha'))$ a *cell type conflict*, or just *type conflict* for short.

We can look at a type conflict from two different perspectives. First, we could regard α as violating the expectation or assumption α' in the formula stored at a'. On the other hand, the reference to a could be wrong or a different function should be used in the formula stored at a' to make correct use of the α value in a. Consequently, a type conflict can point to two different possible sources of errors, that is, the error could either be in a, which means the formula in a' correctly expects a value at a of type α', or the error could be in a', which means the type α for a is correct and the formula in a' contains a wrong function or a wrong reference to a.

Since type conflicts form a many-to-many binary relationship on type cells, we have principally two possibilities of how to represent and report them in a spreadsheet. First, we could group with respect to a, that is, report all cells $(a_1, \alpha_1), \ldots, (a_n, \alpha_n)$ that expect a different type than α for a. This approach amounts to what we could call identifying *downstream type violations*. Alternatively, we could also group with respect to a', that is, report all cells $(a_1, \alpha_1), \ldots, (a_n, \alpha_n)$ that yield a different type than is expected by Γ (assuming that $S(s) : \Gamma \Rightarrow \alpha'$). This approach identifies *upstream type violations*.

Both of these approaches have their merits and could actually be formalized and implemented to give valuable feedback to a spreadsheet user. In the following we focus on upstream type violations since this view supports the identification of template types and the comparison of type-based templates with a notion of templates that is based purely on the syntactic structure of formulas.

An upstream type violation for a cell a' can be represented by a mapping from addresses to type mismatches (that is, pairs of expected and actually found types).

$$V \in \mathcal{V} = A \to T \times T \qquad \textit{upstream type violations}$$

Only pairs of different types should be part of a type violation. The following function eliminates pairs of the form (α,α) from a mapping V. This function will be used in Figure 8.

$$|V| = \{(a, (\alpha, \alpha')) \in V \mid \alpha \neq \alpha'\}$$

Now a *cell type* is given by constant type and a type violation. We use the γ to range over cell types.

$$\gamma \in \mathcal{C} \quad ::= \quad (\alpha, V) \qquad \textit{cell types}$$

The typing rules for cells given in Figure 8 define the judgment $S \triangleright a : \gamma$ that expresses that an individual cell $C_S(a)$ has the cell type γ. The cell typing rule determines the type of a formula f at address a and the cell types (α'_i, V_i) of all cells that are referenced by f. For each referenced cell, a possible type violation is considered by forming the pair (α_i, α'_i) where α_i is the type expected for reference a_i. Application of the function $|\cdot|$ ensures that only true type violations will remain in the constructed cell type for a.

$$\text{UNDEF}_\triangleright \; \frac{a \notin dom(S)}{S \triangleright a : (\mathsf{Undef}, \varnothing)}$$

$$\text{CELL}_\triangleright \; \frac{S(a) : \{\ldots, (\delta_i, \alpha_i), \ldots\} \Rightarrow \alpha \qquad S \triangleright a + \delta_i : (\alpha'_i, V_i)}{S \triangleright a : (\alpha, |\{\ldots, (a + \delta_i, (\alpha_i, \alpha'_i)), \ldots\}|)}$$

Figure 8. Typing rule for cells.

Now a cell (a, f) is said to be *type correct* (in S) if its formula is type correct, and if its type violation is empty, that is, if $S \triangleright a : (\alpha, \varnothing)$.

Note that this definition captures a "local" notion of type correctness for one cell because rule $\text{CELL}_\triangleright$ ignores possible type violations in the referenced cells (we only check each result type α'_i). We could also define the notion of *transitive type correctness*, which requires that in addition to the cell (a, f) all referenced cells also have to be type correct. This

concept could be used to identify connected type correct regions in a spreadsheet. However, these considerations do not contribute to the results of this paper and are therefore not considered any further.

5.4 Typing of Spreadsheets

Spreadsheet types are given by mappings from addresses to cell types.

$$\sigma : A \to \mathcal{C} \qquad \textit{spreadsheet types}$$

The typing rule for spreadsheets is given in Figure 9. It defines the judgment $\triangleright S : \sigma$ expressing that the spreadsheet S has the spreadsheet type σ. The rule determines the cell types for all cells in S and assigns a corresponding spreadsheet type to S.

$$\textsc{Sheet}_{\triangleright}\quad \frac{dom(S) = \{a_1, \ldots, a_n\} \qquad S \triangleright a_i : \gamma_i}{\triangleright S : \{(a_1, \gamma_1), \ldots, (a_n, \gamma_n)\}}$$

Figure 9. Typing rule for spreadsheets.

Now we can see that two cells containing the same formula might differ with regard to their type correctness since their embedding at two different places in a spreadsheet might cause different cells with different types to be referenced.

We can define that two cells are *upstream type equivalent* in a spreadsheet S if they have the same cell types in S.

$$(a_1, f_1) \overset{\gamma}{\leftrightarrow} (a_2, f_2) \iff S \triangleright a_1 : \gamma \wedge S \triangleright a_2 : \gamma$$

Since we do not consider downstream type equivalence any further, we also simply speak of type equivalence when we actually mean upstream type equivalence. We can now reconsider the result of Lemma 1. Even though two cp-similar formulas are type equivalent, the same does generally not hold for two cp-similar cells. Consider, for example, the following spreadsheet.

$$S = \{(\mathsf{A1}, 3), (\mathsf{B1}, \mathsf{True}), (\mathsf{A2}, \mathsf{not\ (0,\text{-}1)}), (\mathsf{B2}, \mathsf{not\ (0,\text{-}1)})\}$$

Although the formulas in A2 and B2 are cp-similar, the cells (A2, not (0,-1)) and (B2, not (0,-1)) are not type equivalent because:

- $S \triangleright \mathsf{A2} : (\mathsf{Bool}, \{(\mathsf{A1}, (\mathsf{Bool}, \mathsf{Num}))\}$ and
- $S \triangleright \mathsf{B2} : (\mathsf{Bool}, \varnothing)$

5.5 Typing of Templates

If we consider a spreadsheet type, we can observe in many cases that large cell areas will have the same type. This will be particularly the case for large, regularly structured spreadsheets (that contain few type errors). The type of these spreadsheets can be described more concisely by condensing areas of repeating types into a constant-size type information. This approach is similar to representing the type of the list [2,3,1,7,4] by a type [Int] instead of Int Int Int Int Int.

Instead of a mapping from addresses to cell types, a spreadsheet type can also be viewed as a sequence of column types where each column is a sequence of cell types. In this representation, a sequence of one and the same type, say Num ... Num can be compressed to {Num} similarly to the list example shown above. Since repetitions in spreadsheets do not only occur on the cell level, but in general on groups of cells, we extend the repeating concept to a group of types.

Therefore, we can view the type of a spreadsheet as a sequence of single column types and repeating groups of column types where a column type is a sequence of single cell types and repeating groups of cell types. This idea is captured in the following abstract grammar.

$$\begin{array}{llll} \kappa & ::= & \gamma \mid \{\gamma \ldots \gamma\} \mid \kappa\kappa & \textit{column types} \\ \tau & ::= & \lceil \kappa \rceil \mid \{\lceil \kappa \rceil \ldots \lceil \kappa \rceil\} \mid \tau\tau & \textit{template types} \end{array}$$

The goal of template types is to provide a succinct, condensed representation of spreadsheet types by identifying repeated groups of types in columns and rows. We introduce some notations to be able to describe the typing rules concisely. First, a sequence of cell types is written as $\bar{\gamma}$. We use the same notation for sequences of column types ($\bar{\kappa}$) and sequences of template types ($\bar{\tau}$). The maximum column in a spreadsheet is obtained by $\overrightarrow{S} = \max\{i \mid (i, j) \in dom(S)\}$. In the definition of the typing rules we use the following abbreviation that allows us to extract the list of cell types for one particular column k.

$$\sigma\lceil k \rceil = \sigma(k, 1) \ldots \sigma(k, n) \quad \text{where} \quad n = \overrightarrow{S}$$

We will later re-use the same notation to refer to all cells in a particular column of a spreadsheet (that is, $S\lceil k \rceil$).

The rules shown in Figure 10 allow to express spreadsheet types as possibly compressed nested sequences of column and cell types, captured by the judgment $\vdash S : \tau$. The first rule $\textsc{Sheet}_{\vdash}$ extracts the column types from the type inferred for a spreadsheet and reformulates it as a template type simply given by the sequence of the columns. The rule $\textsc{Ver}_{\vdash}$ initiates the compression of sequences of cell types: When two identical sequences of cell types $\bar{\gamma}$ that directly follow each other can be found in any column, they can be represented as a repeating group $\{\bar{\gamma}\}$. The rule $\textsc{Ver*}_{\vdash}$ allows the continued reduction of cell sequences into already created repeating groups. The rules $\textsc{Hor}_{\vdash}$ and $\textsc{Hor*}_{\vdash}$ perform the same kind of compression as $\textsc{Ver}_{\vdash}$ and $\textsc{Ver*}_{\vdash}$, only that they allow the grouping of sequences of columns.

$$\textsc{Sheet}_{\vdash}\quad \frac{\triangleright S : \sigma \qquad \sigma\lceil 1 \rceil = \kappa_1 \quad \ldots \quad \sigma\lceil n \rceil = \kappa_n \qquad \overrightarrow{S} = n}{\vdash S : \lceil \kappa_1 \rceil \ldots \lceil \kappa_n \rceil}$$

$$\textsc{Ver}_{\vdash}\quad \frac{\vdash S : \tau_1 \ldots \lceil \bar{\gamma}_1 \bar{\gamma}\bar{\gamma}\bar{\gamma}_2 \rceil \ldots \tau_n}{\vdash S : \tau_1 \ldots \lceil \bar{\gamma}_1 \{\bar{\gamma}\} \bar{\gamma}_2 \rceil \ldots \tau_n} \qquad \textsc{Ver*}_{\vdash}\quad \frac{\vdash S : \tau_1 \ldots \lceil \bar{\gamma}_1 \{\bar{\gamma}\}\bar{\gamma}\bar{\gamma}_2 \rceil \ldots \tau_n}{\vdash S : \tau_1 \ldots \lceil \bar{\gamma}_1 \{\bar{\gamma}\} \bar{\gamma}_2 \rceil \ldots \tau_n}$$

$$\textsc{Hor}_{\vdash}\quad \frac{\vdash S : \bar{\tau}_1 \bar{\tau}\bar{\tau}\bar{\tau}_2}{\vdash S : \bar{\tau}_1 \{\bar{\tau}\} \bar{\tau}_2} \qquad \textsc{Hor*}_{\vdash}\quad \frac{\vdash S : \bar{\tau}_1 \{\bar{\tau}\}\bar{\tau}\bar{\tau}_2}{\vdash S : \bar{\tau}_1 \{\bar{\tau}\} \bar{\tau}_2}$$

Figure 10. Typing rules for templates.

The purpose of template types is to summarize the essential type structure of a spreadsheet by identifying and compressing repeating groups of cell and column types. Therefore, we can expect that many different spreadsheet types can be compressed into one and the same template type. We can formalize this idea by defining, based on the template typing rules, when a spreadsheet type σ is an *instance* of a template type τ.

$$\sigma \prec \tau \iff \exists S \text{ such that } \triangleright S : \sigma \wedge \vdash S : \tau$$

Based on this concept of *template type instance*, we can define a partial ordering on templates to express when a

template τ is *more general* than a template τ'.

$$\tau \sqsubseteq \tau' \iff (\sigma \prec \tau' \Longrightarrow \sigma \prec \tau)$$

6. Type Inference Through Pattern Inference

As discussed in Section 3.2, automatic inference of templates from spreadsheets makes it easier for users to work with systems like Gencel. Templates could also help the users understand their spreadsheets better since templates present a condensed view of potentially large spreadsheets. Spreadsheet templates can be defined and inferred using any equivalence relationship on cells. In the system described in [4], we have used the cp-similarity condition to find similar regions to overlay while inferring templates. Since the process of finding similar regions is basically parametric with respect to the similarity criterion, type equivalence could also be used for inferring templates.

In this section, we present an algorithm to infer a template given some equivalence relationship $\equiv_\eta$ on formulas. This algorithm is a generalization of the algorithm used in [4]. We will revisit the example in Figure 1 to show how different templates would be inferred based on the equivalence criterion used.

The structure of templates can be represented by patterns that can be generated from the following grammar.

$$p_x \quad ::= \quad (f, x) \mid p_x \cdot p_x \mid p_x^n$$

That is, a pattern p_x can be a formula f paired with some additional information of type x, the composition of two patterns, or n repetitions of the same pattern.

The equivalence relationship $\equiv_\eta$ is assumed to be derived from a function $\eta : F \rightarrow x$ that obtains information to be compared from formulas.

$$(a_1, f_1) \equiv_\eta (a_2, f_2) \iff \eta(f_1) = \eta(f_2)$$

The equivalence relationship can be extended to columns, that is, we can say that two columns $S\lceil i \rceil$ and $S\lceil j \rceil$ are $\equiv_\eta$-equivalent if the corresponding cells within the columns are $\equiv_\eta$-equivalent.

$$S\lceil i \rceil \equiv_\eta S\lceil j \rceil \iff \forall k : C_S(i,k) \equiv_\eta C_S(j,k)$$

We can extend $\equiv_\eta$ one step further to regions within spreadsheets. A horizontal region can be formed from the repeated horizontal composition of adjacent columns of cells. We say that two regions are $\equiv_\eta$-equivalent if they both have the same number of columns and the corresponding columns are $\equiv_\eta$-equivalent. The importance of identifying equivalent regions lies in the fact that horizontally aligned equivalent regions can be *compressed* to a single instance in the template by repeatedly overlaying the equivalent regions onto the innermost copy.

The main steps involved in inferring a pattern from a given spreadsheet are shown in the definition of the function PatGen(S). First, the function Infer($S, \equiv_\eta$) extends all cells in the spreadsheet S by the result of the function η, on which the equivalence relationship $\equiv_\eta$ is based. Having the information on which the equivalence-class computations are based explicitly available eventually facilitates the generation of template types from patterns.

Next, the function PatGen is called on the extended spreadsheet $\tilde{S}$. This approach also means that the equivalence comparisons can all be performed by the function "$=_\eta$", which simply compares the second component of the second component of a pair.

$$(a_1, (f_1, x_1)) =_\eta (a_2, (f_2, x_2)) \iff x_1 = x_2$$

Since $=_\eta$ is essentially a polymorphic function, it doesn't have to be passed as a parameter to algorithms.

Algorithm 1: Pattern Inference
Input: Spreadsheet S and the equivalence relation $\equiv_\eta$.
Output: Template for the spreadsheet S.

```
INFER(S, ≡η)
(1)  S̃ ← {(a, (f, x)) | (a, f) ∈ S, x = η(f)}
(2)  S̃' ← PATGEN(S̃)
(3)  return (S̃')
```

Algorithm 2: Pattern Generation
Input: Spreadsheet S.
Output: Pattern p_x (which is S' with the number of repetitions marked).

```
PATGEN(S)
(1)   G ← S/=η
(2)   S' ← S
(3)   while G ≠ ∅
(4)       foreach g ∈ G
(5)           Sp ← MAXOVERLAY(g)
(6)           if Sp = () then continue
(7)           else
(8)               (S⌈i⌉, S⌈j⌉) ← Sp
(9)               S'⌈i⌉ ← OVERLAY(i, j)
(10)      if S = S' then return S
(11)      else
(12)          G ← S'/=η
(13)          S ← S'
(14)  return S'
```

To infer the template for a given spreadsheet, we first partition columns into $\equiv_\eta$ equivalence classes, that is, we compute $S/{=_\eta}$. For each group g of columns in an equivalence class, we determine the columns that we can overlay that would in turn result in the biggest regions being overlaid by the call to MaxOverlay(g). If an overlay is not possible for the group under consideration, we pick the next group, and the overlays are carried out whenever possible. Note that every time one region is overlaid on another, we increment a counter at the column level to keep track of the number of times a column instance has been compressed. This information is stored in S'. Therefore, the call to PatGen(S) returns S' along with the number of repetitions of the columns in S', that is p_x.

MaxOverlay(g) takes an equivalence group g of columns and returns the indices of the two columns in g that are furthest apart and can be overlaid. For an overlay to be successful, it must meet the following condition

$$\forall k \in \{0, ..., d\} : S\lceil i - k \rceil \equiv_\eta S\lceil j - k \rceil$$

where $d = j - i - 1$. That is, $S\lceil j \rceil$ can be overlaid on $S\lceil i \rceil$ if and only if the columns between $S\lceil i \rceil$ and $S\lceil j \rceil$ can be overlaid on the corresponding columns before $S\lceil i \rceil$. If no overlay is possible for g, MaxOverlay(g) returns the null tuple.

Overlay(i, j) takes the two columns that have to be overlaid and returns the resulting sheet S''. Before overlaying one column over another, we first need to update the formulas in the cells in the region $S\lceil j+1..j' \rceil$ by the call to UpdateFormula($S\lceil j+1..j' \rceil, i+1, j$) to remove references to cells within region $S\lceil i+1..j \rceil$. This transformation

Algorithm 3: Maximal overlay in an equivalence group.
Input: Equivalence class g.
Output: The column numbers l and m that result in the maximal overlay.

MAXOVERLAY(g)
(1) $d_m \leftarrow 0$
(2) $l \leftarrow 0$
(3) $m \leftarrow 0$
(4) **foreach** $S\lceil i \rceil \in g$
(5) **foreach** $S\lceil j \rceil \in g \wedge j \neq i$
(6) **if** $j > i$ **then** $d \leftarrow j - i$
(7) **else** $d \leftarrow i - j$
(8) **if** $\forall k \in \{0, ..., d\} : S\lceil i - k \rceil =_\eta S\lceil j - k \rceil$
(9) **if** $d > d_m$
(10) $d_m \leftarrow d$
(11) $l \leftarrow \min(i, j)$
(12) $m \leftarrow \max(i, j)$
(13) **if** $l = 0 \vee m = 0$ **then return** ()
(14) **else return** (l, m)

of the formulas is subject to the constraint that references to cells within repeated groups from outside the repeated group have to be in aggregation formulas because we do not know beforehand how many times the region might be repeated. Since this update can only be done on aggregation formulas, overlaying fails (and will not be performed) if some formula in the region $S\lceil j + 1..j' \rceil$ cannot be updated.

Algorithm 4: Overlaying two columns from the same equivalence class.
Input: Spreadsheet S and column numbers i and j where $i < j$.
Output: Resulting spreadsheet S' after overlay.

OVERLAY(S, i, j)
(1) $j' \leftarrow \overrightarrow{S}$
(2) $d \leftarrow j - i$
(3) $S' \leftarrow$ UPDATEFORMULA($S\lceil j + 1..j' \rceil, i + 1, j$)
(4) $S'' \leftarrow$ SHIFT($S', j + 1, j', d$)
(5) **return** S''

Finally, we need to shift the cells in region $S\lceil n + 1..m \rceil$ to reflect the overlay, that is, each cell in the updated spreadsheet S' is mapped to a cell in the original spreadsheet S by SHIFT(S, m, n, d).

Algorithm 5: Adjusting columns after overlay
Input: Spreadsheet S, the start (m) and end (n) positions of the columns to be adjusted, and the width (d) of the region being overlaid.
Output: Resulting spreadsheet S' after column adjustment.

SHIFT(S, m, n, d)
(1) **for** $i = 1$ **to** $m - d - 1$
(2) $S'\lceil i \rceil = S\lceil i \rceil$
(3) **for** $i = m - d$ **to** $n - d$
(4) $S'\lceil i \rceil = S\lceil i + d \rceil$
(5) **return** S'

We repeat this process until there are no more $\equiv_\eta$-equivalent columns that can be overlaid and we are left with the compressed sheet S'. We then carry out a similar process and compress the spreadsheet S' vertically by considering the $\equiv_\eta$-equivalent rows that can be overlaid. We are left with the pattern p_x which is the compressed form of the original spreadsheet S.

When we call INFER with the equivalence relationship $\overset{\gamma}{\leftrightarrow}$, which is based on the cell types determined by the type inference judgment $S \triangleright a : \gamma$, the generated pattern p_γ carries cell types and represents a template type that can be extracted by a two-level traversal.

Since the grouping of repeating columns and rows is based on taking maximal repeating blocks, it follows from the definition of template type instance given in Section 5.5 that pattern inference produces template types that are at least as general as those that can be inferred by the typing rules.

THEOREM 1 (Correctness of Type Inference).

$$\vdash S : \tau' \wedge \text{INFER}(S, \overset{\gamma}{\leftrightarrow}) = \tau \Longrightarrow \tau \sqsubseteq \tau'$$

In the following section we will compare the type inference algorithm with the template inference based on cp-similarity.

7. Comparison of Template Inferences

As mentioned earlier in the paper, cp-similarity has been exploited in consistency checking [27] and testing of spreadsheets [10]. However, type similarity has never been used before. Since formulas with varying levels of similarities occur frequently in spreadsheets as a result of the repetitive actions (for example, copy-paste, click-and-drag, etc.) employed by spreadsheet programmers, violations of these similarities can be indicative of faults. The frequency of occurrence of cp-similar regions has been shown by the analyses carried out on the EUSES spreadsheet corpus as reported in [18]. The corpus has 4498 spreadsheets collected from various sources. Out of the 1977 spreadsheets in the corpus that have formulas in them, 1797 have cp-similar regions. Among the sheets that have cp-similar regions, there are on average 5.2 regions per sheet, with an average of 13.1 regions in spreadsheets that had at least 1 region, a maximum of 414 regions in a spreadsheet, and 23845 regions in total in all the spreadsheets.

The system described in [4] infers templates from spreadsheets by overlaying cp-similar regions. When this system is run on the grade spreadsheet shown in Figure 1, it is unable to do any overlay because of the errors in the sheet.

1. Row-level overlays fail because the formulas in I3, I4, I5, and I6 cannot be grouped using the cp-similarity condition.
2. The columns with the scores for the assignments are potential candidates for column-level overlay. However, the formulas in columns C, E, and G fail to satisfy the cp-similarity condition because of the errors in some of the cells.

Expecting the grade spreadsheet formulas to be the same for the different students, the user might invoke the system to infer the template, hoping it generates a template along the lines of the one shown in Figure 5. The failure of the system to do any compression at all is indicative of the faults present in the spreadsheet.

We assume the user corrects the errors in columns E, G, H, I, and J and runs the template inference system again. This particular scenario helps to illustrate how the template inference algorithm based on cp-similarity works. This will also enable the comparison with the type inference algorithm.

When we use cp-similarity as the equivalence relationship $\equiv_\eta$, assuming we already have only relative references in the spreadsheet formulas, η is simply the identity function.

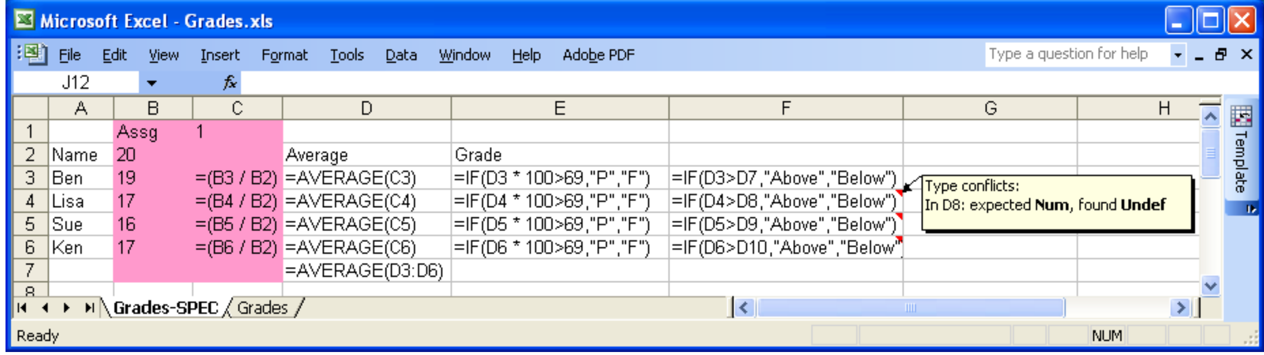


Figure 11. Type-equivalence-based template for the grade sheet.

Therefore, the call to INFER$(S, \equiv_\eta)$ in the first step generates the following extended spreadsheet.

$$\tilde{S} \leftarrow \{(a, (f, f)) \mid (a, f) \in S\}$$

The extended spreadsheet is passed to PATGEN(S) and the result is returned by the outermost function INFER.

In the first step in PATGEN, the columns are partitioned into equivalence classes on the basis of cp-similarity. For the sake of conciseness we represent columns by their column numbers.

$$G \leftarrow \{\{3, 5, 7\}\}$$

Note that we find only one group in G. To decide the best overlay for this group of columns, MAXOVERLAY is called on the group. Even though $d = 4$ when $i = 3$ and $j = 7$, the overlay is not possible since the following condition is not satisfied.

$$\forall k \in \{0, ..., 4\} : S\lceil 3 - k \rceil =_\eta S\lceil 7 - k \rceil$$

For $(i = 3, j = 5)$ and $(i = 5, j = 7)$ we both have $d = 2$. The first case is slected, and MAXOVERLAY returns $(3, 5)$.

The call to OVERLAY$(S, 3, 5)$, in turn calls UPDATEFORMULA$(S\lceil 6..10 \rceil, 4, 5)$ which updates the formulas in columns by removing references to cells in columns 4 and 5. For example, the formula in H6 becomes AVERAGE(C6,G6) after the update. After the formulas have been updated, the call to SHIFT$(S, 6, 10, 2)$ copies the columns 1 through 3, and columns 6 through 10 to the resulting spreadsheet, which is then returned by the function.

After the first overlay has been carried out, the columns in the resulting spreadsheet can once again be partitioned into equivalence classes on the basis of cp-similarity.

$$G \leftarrow \{\{3, 5\}\}$$

In this case, column 5 refers to column 7 in the original spreadsheet. Once again, the overlay can be carried out, after which the partitioning does not generate any more candidates for overlaying. This indicates that the spreadsheet has been compressed horizontally as much as possible.

The system then proceeds to compress the rows in the resulting spreadsheet, carrying out the following overlays one after another: $(3, 4)$, $(3, 4)$, and again $(3, 4)$, which means the system first overlays row 4 on row 3. In the resulting spreadsheet, the system once again overlays the new row 4 on row 3. After the same overlay is performed a third time, the result will be the template shown in Figure 5.

Since cell type equivalence is a stronger condition than cp-similarity of formulas, using type-equivalence for inferring the templates helps detect errors that would not be detected by the use of formula cp-similarity. For example, assume the errors in the spreadsheet formulas, except those in column J, have been corrected. Now if we use cp-similarity as the equivalence condition for inferring the template, the system infers the template shown in Figure 5. While the template can be used in case the user wants to continue working within the ViTSL/Gencel framework, the errors in the formulas in column J would be still present in the spreadsheet. In contrast, when template inference is carried out using cell type equivalence, columns E and G will be overlaid on C. However, overlaying of rows fails because the formulas in J3, J4, J5, and J6 are cp-similar but not type equivalent since they have the following types.

- $S \triangleright$ J3 : (String, $\varnothing$)
- $S \triangleright$ J4 : (String, {(H8, (Num, Undef))})
- $S \triangleright$ J5 : (String, {(H9, (Num, Undef))})
- $S \triangleright$ J6 : (String, {(H10, (Num, Undef))})

More precisely, using the type inference algorithm, the template shown in Figure 11 would be produced.

The failure of the expected row-level overlay and the difference in type expectations of the cells can point the user to the faults in the spreadsheet. After correcting those errors, a repeated template inference attempt would yield the expected template.

8. Conclusions and Future Research

In this paper we have presented a type system that characterizes types of formulas and cells on a fine-grained level to allow detailed reports about errors in spreadsheet cells. In addition, the definition of template types allows the concise description of spreadsheet types, which is particularly beneficial in large spreadsheets by providing summaries of the spreadsheets' type structures.

We have also presented a type inference algorithm that is based on a generic pattern inference algorithm to identify repeated, similar areas in spreadsheets based on different equivalence relationships. The type inference algorithm is obtained by instantiating pattern inference by a notion of type equivalence that is based on the type system we have introduced. We have demonstrated that the type-equivalence-based inference leads to patterns that provide more accurate models of spreadsheets than produced by purely syntactic approaches.

In addition to the applications considered here, there are more aspects of the type system that can be investigated.

We have indicated that the notion of *downstream type equivalence* can provide an alternative viewpoint of type conflicts. One particular aspect is "*voting for type errors*" by counting the number of references of expected type α for a cell that contains a value of type $\alpha' \neq \alpha$. Numbers greater than one can be taken as an indication that the error is more likely in the value or formula contained in the referenced cell than in the referencing formulas. *Transitive type analysis*, as indicated in Section 5.3, is another area that might reveal interesting opportunities for new forms of templates.

References

[1] R. Abraham and M. Erwig. Header and Unit Inference for Spreadsheets Through Spatial Analyses. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 165–172, 2004.

[2] R. Abraham and M. Erwig. Goal-Directed Debugging of Spreadsheets. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 37–44, 2005.

[3] R. Abraham and M. Erwig. How to Communicate Unit Error Messages in Spreadsheets. In *1st Workshop on End-User Software Engineering*, pages 52–56, 2005.

[4] R. Abraham and M. Erwig. Inferring Templates from Spreadsheets. In *28th IEEE Int. Conf. on Software Engineering*, 2006. To appear.

[5] R. Abraham, M. Erwig, S. Kollmansberger, and E. Seifert. Visual Specifications of Correct Spreadsheets. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 189–196, 2005.

[6] Y. Ahmad, T. Antoniu, S. Goldwater, and S. Krishnamurthi. A Type System for Statically Detecting Spreadsheet Errors. In *18th IEEE Int. Conf. on Automated Software Engineering*, pages 174–183, 2003.

[7] T. Antoniu, P. A. Steckler, S. Krishnamurthi, E. Neuwirth, and M. Felleisen. Validating the Unit Correctness of Spreadsheet Programs. In *26th IEEE Int. Conf. on Software Engineering*, pages 439–448, 2004.

[8] P. S. Brown and J. D. Gould. An Experimental Study of People Creating Spreadsheets. *ACM Transactions on Office Information Systems*, 5(3):258–272, 1987.

[9] M. M. Burnett, C. Cook, J. Summet, G. Rothermel, and C. Wallace. End-User Software Engineering with Assertions. In *25th IEEE Int. Conf. on Software Engineering*, pages 93–103, 2003.

[10] M. M. Burnett, A. Sheretov, B. Ren, and G. Rothermel. Testing Homogeneous Spreadsheet Grids with the "What You See Is What You Test" Methodology. *IEEE Transactions on Software Engineering*, 29(6):576–594, 2002.

[11] M. J. Coblenz, A. J. Ko, and B. A. Myers. Using Objects of Measurement to Detect Spreadsheet Errors. In *IEEE Int. Symp. on Visual Languages and Human-Centric Computing*, pages 314–316, 2005.

[12] A. Ebrahimi. Novice Programmer Errors: Language Constructs and Plan Composition. *Int. Journal of Human-Computer Studies*, 41(4):457–480, 1994.

[13] G. Engels and M. Erwig. ClassSheets: Automatic Generation of Spreadsheet Applications from Object-Oriented Specifications. In *20th IEEE/ACM Int. Conf. on Automated Software Engineering*, pages 124–133, 2005.

[14] M. Erwig, R. Abraham, I. Cooperstein, and S. Kollmansberger. Automatic Generation and Maintenance of Correct Spreadsheets. In *27th IEEE Int. Conf. on Software Engineering*, pages 136–145, 2005.

[15] M. Erwig, R. Abraham, S. Kollmansberger, and I. Cooperstein. Gencel — A Program Generator for Correct Spreadsheets. *Journal of Functional Programming*, 16(3):293–325, 2006.

[16] 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.

[17] EuSpRIG. European Spreadsheet Risks Interest Group. `http://www.eusprig.org/`.

[18] M. Fisher and G. Rothermel. The EUSES Spreadsheet Corpus: A Shared Resource for Supporting Experimentation with Spreadsheet Dependability Mechanism. In *1st Workshop on End-User Software Engineering*, pages 47–51, 2005.

[19] M. Fisher II, M. Cao, G. Rothermel, C. Cook, and M. M. Burnett. Automated Test Case Generation for Spreadsheets. In *24th IEEE Int. Conf. on Software Engineering*, pages 141–151, 2002.

[20] J. D. Gannon. An Experimental Evaluation of Data Type Conventions. *Communications of the ACM*, 20(8):584–595, 1977.

[21] K. Godfrey. Computing Error at Fidelity's Magellan Fund. *The Risks Digest*, 16(72), 1995.

[22] D. G. Hendry and T. R. G. Green. Creating, Comprehending and Explaining Spreadsheets: A Cognitive Interpretation of What Discretionary Users Think of the Spreadsheet Model. *International Journal of Human-Computer Studies* , 40:1033–1065, 1994.

[23] T. Isakowitz, S. Schocken, and H. C. Lucas, Jr. Toward a Logical/Physical Theory of Spreadsheet Modelling. *ACM Transactions on Information Systems*, 13(1):1–37, 1995.

[24] A. Kay. Computer Software. *Scientific American*, 251(3):41–47, 1984.

[25] J. F. Lerch, M. M. Mantei, and J. R. Olson. Skilled Financial Planning: The Cost of Translating Ideas Into Action. *ACM Conf. on Human Factors in Computing Systems*, pages 121–126, 1989.

[26] C. Lewis and G. M. Olson. Can Principles of Cognition Lower the Barriers to Programming? In *2nd Workshop on Empirical Studies of Programmers*, pages 248–263, 1987.

[27] R. Mittermeir and M. Clermont. Finding High-Level Structures in Spreadsheet Programs. In *9th Working Conference on Reverse Engineering*, pages 221–232, 2002.

[28] D. A. Norman. Cognitive Engineering. In D. A. Norman and S. W. Draper, editors, *User-Centered System Design*, pages 31–61. Hillsale, NJ: Lawrence Erlbaum, 1986.

[29] R. R. Panko. What We Know About Spreadsheet Errors. *Journal of End User Computing (Special issue on Scaling Up End User Development)*, 10(2):15–21, 1998.

[30] R. R. Panko. Applying Code Inspection to Spreadsheet Testing. *Journal of Management Information Systems*, 16(2):159–176, 1999.

[31] R. R. Panko. Spreadsheet Errors: What We Know. What We Think We Can Do. In *Symp. of the European Spreadsheet Risks Interest Group (EuSpRIG)*, 2000.

[32] S. L. Peyton Jones, A. Blackwell, and M. M. Burnett. A User-Centered Approach to Functions in Excel. In *ACM Int. Conf. on Functional Programming*, pages 165–176, 2003.

[33] L. Prechelt and W. F. Tichy. A Controlled Experiment to Assess the Benefits of Procedure Argument Type Checking. *IEEE Transactions on Software Engineering*, 24(4):302–312, 1998.

[34] K. Rajalingham, D. Chadwick, B. Knight, and D. Edwards. Quality Control in Spreadsheets: A Software Engineering-Based Approach to Spreadsheet Development. In *33rd Hawaii Int. Conf. on System Sciences*, pages 1–9, 2000.

[35] K. Rajalingham, D. R. Chadwick, and B. Knight. Classification of Spreadsheet Errors. *Symp. of the European Spreadsheet Risks Interest Group (EuSpRIG)*, 2001.

[36] B. Ronen, M. A. Palley, and H. C. Lucas, Jr. Spreadsheet Analysis and Design. *Communications of the ACM*, 32(1):84–93, 1989.

[37] G. Rothermel, M. M. Burnett, L. Li, C. DuPuis, and A. Sheretov. A Methodology for Testing Spreadsheets. *ACM Transactions on Software Engineering and Methodology*, pages 110–147, 2001.

[38] J. Sajaniemi. Modeling Spreadsheet Audit: A Rigorous Approach to Automatic Visualization. *Journal of Visual Languages and Computing*, 11:49–82, 2000.

[39] C. Scaffidi, M. Shaw, and B. Myers. Estimating the Numbers of End Users and End User Programmers. In *IEEE Symp. on Visual Languages and Human-Centric Computing*, pages 207–214.

[40] J. C. Spohrer and E. Soloway. Novice Mistakes: Are the Folk Wisdoms Correct? *Communications of the ACM*, 29(7):624–632, 1986.

[41] J. G. Spohrer and E. Soloway. Analyzing the High Frequency Bugs in Novice Programs. In *First Workshop on Empirical Studies of Programmers*, pages 230–251, 1986.

[42] A. G. Yoder and D. L. Cohn. Real Spreadsheets for Real Programmers. In *Int. Conf. on Computer Languages*, pages 20–30, 1994.