Designing a strongly typed spreadsheet

Brad Neimann

August 23, 2023

‘Vernacular software’

myth	*se mythos*	*se pragmos*
Professional Programmer	Programs are written by highly skilled professional programmers.	Vernacular software developers vastly outnumber professional developers. Professional developers now (mostly) do things other than write code.
The Code IS the Software	Software is simply the symbolic program text.	Software systems are coalitions of many types of elements from many sources with sketchy specifications and unannounced behavior change.
Mathematical Tractability	Soundness of programming languages is essential.	Task-specific expressiveness is more important than completeness or soundness.
Correctness	Correctness of software is also essential.	Fitness for task usually matters more than absolute correctness.
Specifications	Thus formal specifications are also essential.	Much software is developed to discover what it should do, not to satisfy a prior specification.

Mary Shaw. “Myths and Mythconceptions: What Does It Mean to Be a Programming Language, Anyhow?” In: *Proc. ACM Program. Lang.* 4.HOPL (Apr. 2022), 234:1–234:44. DOI: 10.1145/3480947. (Visited on 02/20/2023)

But correctness does cause issues!

> “… *each template could handle only about 65,000 rows of data rather than the one million-plus rows that Excel is actually capable of.*
>
> *And since each test result created several rows of data, in practice it meant that each template was limited to about 1,400 cases.*
>
> *When that total was reached, further cases were simply left off.*”

Leo Kelion. “Excel: Why Using Microsoft’s Tool Caused Covid-19 Results to Be Lost”. In: *BBC News* (Oct. 2020). (Visited on 08/17/2023)

But correctness does cause issues!

> “ … *approximately one-fifth of papers with supplementary Excel gene lists contain erroneous gene name conversions.* ”

> “ … *all symbols that autoconverted to dates in Microsoft Excel have been changed (for example, SEPT1 is now SEPTIN1; MARCH1 is now MARCHF1) …* ”

Mark Ziemann, Yotam Eren, and Assam El-Osta. “Gene Name Errors Are Widespread in the Scientific Literature”. In: *Genome Biology* 17.1 (Aug. 2016), p. 177. ISSN: 1474-760X. DOI: `10.1186/s13059-016-1044-7`. (Visited on 08/17/2023)

Elspeth A. Bruford et al. “Guidelines for Human Gene Nomenclature”. In: *Nat Genet* 52.8 (Aug. 2020), pp. 754–758. ISSN: 1546-1718. DOI: `10.1038/s41588-020-0669-3`. (Visited on 08/17/2023)

Spreadsheets: hotbeds of incorrectness

Almost one in two spreadsheets show impactful errors.

> “ *The major symptoms we observed of poor spreadsheet practice are the following:*
>
> - *Chaotic design*
> - *Embedded numbers*
> - *Special cases*
> - *Non-repeating structures*
> - *Complex formulas.* ”

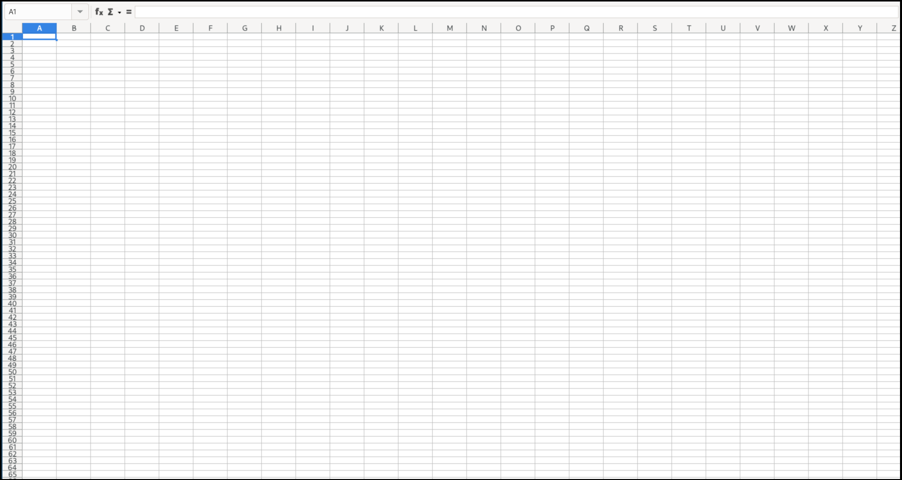
Stephen G. Powell, Barry Lawson, and Kenneth R. Baker. *Impact of Errors in Operational Spreadsheets*. Comment: 12 pages including references. Jan. 2008. DOI: 10.48550/arXiv.0801.0715. arXiv: 0801.0715 [cs]. (Visited on 08/19/2023)

Spreadsheets: hotbeds of incorrectness

The underlying cause?

Spreadsheets are unstructured,
with almost nonexistent abstraction facilities.

Structurelessness: the big grid of cells



Spreadsheets are functional programming languages!

```
=RIGHT(A1,LEN(A1)-FIND("|",SUBSTITUTE(A1," ","|",
     LEN(A1)-LEN(SUBSTITUTE(A1," ","")))))
```

```
=SUMPRODUCT(((
    COUNTIF(OFFSET(A1,ROW(A2:A33)-1,0),"*apple*")+
    COUNTIF(OFFSET(A1,ROW(A2:A33)-1,0),"*seed*")+
    COUNTIF(OFFSET(A1,ROW(A2:A33)-1,0),"*turf*"))>0)
    *(B2:B33="B"))
```


BradC. *Answer to "How Can I Perform a Reverse String Search in Excel without Using VBA?"*. Dec. 2008. (Visited on 08/19/2023)

kasparg. *Answer to "How to Create Excel Complicated Formula for Multiple and/or Criteria"*. June 2018. (Visited on 08/19/2023)

Structured spreadsheets: freedom from the big grid of cells

Want a ‘spreadsheet’ with:

- Proper data structures
- User-defined functions
- Strong type system

Structured spreadsheets: freedom from the big grid of cells

Issue: spreadsheets are interactive!

Our type system needs to be strong,
yet flexible in the face of real-world data
and non-programmer usage.

Are static types really necessary?

After all, the user immediately sees type errors either way.

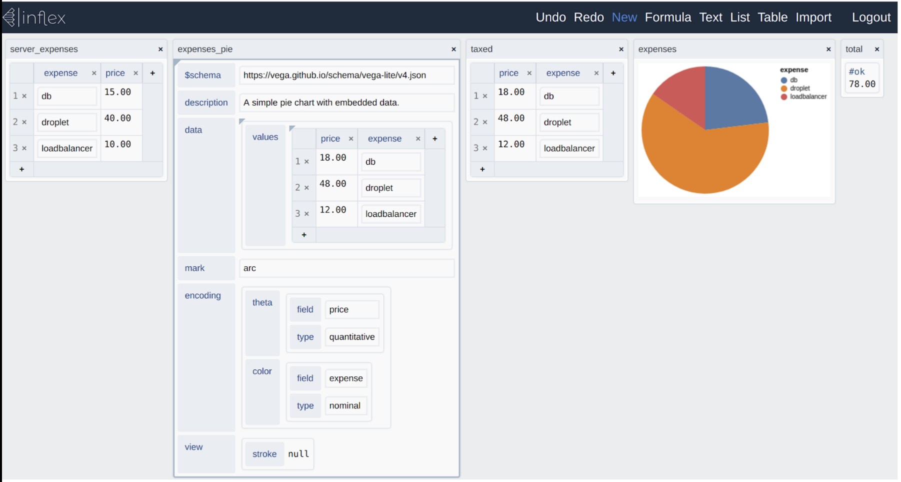
But static typechecking can verify the complete absence of problems, even if that control flow path is not evaluated.

ML-style type systems

Defining properties:

- Admit full, global type inference
- Parametric polymorphism
- Strong abstraction capabilities:
  typeclasses or module functors
- ADTs for defining new types

Example: Inflex



Chris Done. *Inflex*. Aug. 2023. (Visited on 08/19/2023)

Example: Inflex

functions

```
map      :: (a -> b) -> [a] -> [b]
filter   :: (a -> <#true|#false>) -> [a] -> [a]
sum      :: Addable a => [a] -> <#ok(a)|#sum_empty>
average  :: Addable a, Divisible a => [a] -> <#ok(a)|#average_empty>
vega     :: a -> VegaChart
null     :: [a] -> <#true|#false>
length   :: FromInteger number => [a] -> number
distinct :: Comparable a => [a] -> [a]
minimum  :: Comparable a => [a] -> <#ok(a)|#minimum_empty>
maximum  :: Comparable a => [a] -> <#ok(a)|#maximum_empty>
sort     :: Comparable a => [a] -> [a]
find     :: (a -> <#true|#false>) -> [a] -> <#find_empty|#find_failed|#ok(a)>
all      :: (a -> <#true|#false>) -> [a] -> <#all_empty|#ok(a)>
any      :: (a -> <#true|#false>) -> [a] -> <#any_empty|#ok(a)>
from_ok  :: a -> <#ok(a)|v> -> a
```

Chris Done. *Inflex*. Aug. 2023. (Visited on 08/19/2023)

But how do you cope with...

- Categorical data?
- Blank cells?
- Inhomogeneous columns?
- Type conversions?
- Data exploration?

Answer: structural subtyping!

Introduce subtyping:
$T \preceq U$ when T can be used anywhere that U can.

Combining types structurally

- Unions: $T \vee U$
- Intersections: $T \wedge U$
- Negations: $\neg T$
- Records: $\{\texttt{field1}\colon T, \texttt{field2}\colon U\} \leq \{\texttt{field1}\colon T\}$
- Singletons: `0`, `True`, `"string"`, etc.

Combining types structurally

- A list of possibly blank integers: `Int ∨ Blank`
- An enumeration:
  `"igneous" ∨ "sedimentary" ∨ "metamorphic"`
- ADTs:
  `({type: "circle"} ∧ {radius: Int}) ∨`
  `({type: "square"} ∧ {width: Int} ∧ {height: Int})`

Some structurally typed languages

- TypeScript
- Flow
- Sorbet
- Elixir (soon!)

Giuseppe Castagna, Guillaume Duboc, and José Valim. "The Design Principles of the Elixir Type System". In: *The Art, Science, and Engineering of Programming* ()

Structural type inference

But global type inference is in general undecidable for a structural type system!

(even intersection types are too much...)

Benjamin C. Pierce. *Types and Programming Languages*. Cambridge, Mass: MIT Press, 2002. ISBN: 978-0-262-16209-8. (Visited on 08/21/2023)

Structural type inference

With some restrictions it is possible, but very difficult...

MLstruct online demonstration

```
class None: {}
class Some[A]: { value: A }
type Option[A] = Some[A] | None


def flatMap f opt = case opt of
  Some -> f opt.value,
  None -> None{}

res = flatMap (fun x -> x) (Some{value = 42})

case res of int -> res, None -> 0


def flatMap2 f opt =
  case opt of Some -> f opt.value, _ -> opt

ex1 = flatMap2 (fun x -> x) 42

ex2 = flatMap2 (fun x -> Some{value = x}) (Some{value = 12})

ex3 = flatMap2 (fun x -> Some{value = x}) 42


def mapSome f opt =
  case opt of Some -> f opt, _ -> opt


class SomeAnd[A, P]: Some[A] & { payload: P }

let arg =
  if true then SomeAnd{value = 42; payload = 23} else None{}
  in mapSome (fun x -> x.value + x.payload) arg


class Cons[A]: Some[A] & { tail: List[A] }
```

<table>
<tr><th>Name</th><th>Type</th><th>Value</th></tr>
<tr><td>flatMap</td><td>('value -> 'a) -> (None | #Some & {value: 'value}) -> (None | 'a)</td><td>[Function: flatMap]</td></tr>
<tr><td>res</td><td>42 | None</td><td>42</td></tr>
<tr><td>res</td><td>0 | 42</td><td>42</td></tr>
<tr><td>flatMap2</td><td>('value -> 'a) -> (#Some & {value: 'value} | 'a & ~#Some) -> 'a</td><td>[Function: flatMap2]</td></tr>
<tr><td>ex1</td><td>42</td><td>42</td></tr>
<tr><td>ex2</td><td>Some[12]</td><td>Some { "value": 12 }</td></tr>
<tr><td>ex3</td><td>42 | Some[nothing]</td><td>42</td></tr>
<tr><td>mapSome</td><td>('a -> 'b) -> (#Some & 'a | 'b & ~#Some) -> 'b</td><td>[Function: mapSome]</td></tr>
<tr><td>res</td><td>None | int</td><td>65</td></tr>
<tr><td>mapList</td><td>('value -> 'value0) -> 'a -> 'tail
where
'tail :> Cons['value0] & {tail: 'tail} | None
'a <: #Cons & {tail: 'a, value: 'value} | None</td><td>[Function: mapList]</td></tr>
<tr><td>Cons</td><td>('value & 'A) -> (List['A] & 'tail) -> (Cons['A] & {tail: 'tail, value: 'value})</td><td>[Function: Cons1]</td></tr>
<tr><td>None</td><td>None</td><td>None {}</td></tr>
<tr><td>unzip</td><td>'a -> {fst: 'tail, snd: 'tail0}
where
'tail0 :> Cons['value] & {tail: 'tail0} | None
'tail :> Cons['value0] & {tail: 'tail} | None
'a <: None | #Some & {tail: 'a, value: {fst: 'value0, snd: 'value}}</td><td>[Function: unzip]</td></tr>
<tr><td>Cons_tpd</td><td>'a -> (List['a] & 'b) -> (Cons['a] & {tail: 'b})</td><td>no value</td></tr>
</table>

Lionel Parreaux. "The Simple Essence of Algebraic Subtyping: Principal Type Inference with Subtyping Made Easy (Functional Pearl)". In: *Proc. ACM Program. Lang.* 4.ICFP (Aug. 2020). Extends (Dolan) — also contains and explains reference inference implementation in Scala!, 124:1–124:28. DOI: 10.1145/3409006. (Visited on 01/07/2023)

Lionel Parreaux and Chun Yin Chau. "MLstruct: Principal Type Inference in a Boolean Algebra of Structural Types". In: *Proc. ACM Program. Lang.* 6.OOPSLA2 (Oct. 2022), pp. 449–478. ISSN: 2475-1421. DOI: 10.1145/3563304. (Visited on 01/09/2023)

A more manageable approach: local inference

Instead: if top-level function parameters are annotated, everything else can be inferred.

Benjamin C. Pierce and David N. Turner. "Local Type Inference". In: *ACM Trans. Program. Lang. Syst.* 22.1 (Jan. 2000), pp. 1–44. ISSN: 0164-0925. DOI: 10.1145/345099.345100. (Visited on 12/27/2022)

Martin Odersky, Christoph Zenger, and Matthias Zenger. "Colored Local Type Inference". In: *SIGPLAN Not.* 36.3 (Jan. 2001), pp. 41–53. ISSN: 0362-1340. DOI: 10.1145/373243.360207. (Visited on 02/19/2023)

A possible workflow

List1
Int ∨ Blank
1
2
#TERR5
3
4

Difficulties: tabular data

How do you handle table manipulations?

e.g. merging: `merge` $: \forall R_1, R_2.\ (R_1, R_2) \to R_1 \wedge R_2$

Difficulties: most intersections make no sense!

What does it mean to have an $\texttt{Int} \wedge \texttt{Text}$?

Or a $\{\texttt{foo}\colon T\} \wedge \{\texttt{foo}\colon U\}$?

Intersections are only really useful for two things:
records and function overloading
(and we don't really need overloading)

My design: basics

Suggestion: use unions, but not intersections
(or negations, they're confusing too)

Instead, use row types without subtyping relation-ships:

$$\{\texttt{field1} : T, \texttt{field2} : U \mid \rho\}$$

My design: row types

Can also introduce scoping:
$\{\texttt{field1}\colon T, \texttt{field1}\colon U\} \equiv \{\texttt{field1}\colon T\}$

and first-class labels: $(\!|\, l \,|\!)$

Now we can type many functions!

$$\texttt{Get}\colon \forall l, \rho, T.\ ((\!|\, l \,|\!), \{l\colon T, \rho\}) \longrightarrow T$$
$$\texttt{Merge}\colon \forall \rho_1, \rho_2.\ (\{\rho_1\}, \{\rho_2\}) \longrightarrow \{\rho_1, \rho_2\}$$

Adam Paszke and Ningning Xie. "Infix-Extensible Record Types for Tabular Data". In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Type-Driven Development (TyDe '23)*. Seattle, WA, USA: ACM, New York, NY, USA, Sept. 2023. DOI: `10.1145/3609027.3609406`

My design: syntactic choices

Remember: this is aimed at non-programmers!

	Old	New
Union	$T \vee U$	`T or U`
Type variable	t	`?t`
Quantification	$\forall t, u. \ldots$	`<?t,?u> …`
Label	`field1`	`#field1`
Record	$\{\texttt{field1}\colon T, \texttt{field2}\colon U\}$	`{#field1 T, #field2 U}`

For example:

```
Get: <?label, ?t> (?label, {?label ?t}) -> ?t
Merge: <?r1, ?r2> ({?r1}, {?r2}) -> {?r1, ?r2}
```

My design: other goodies

- Units: `1m: Num<m>`
  - Unit conversions: `1m+2cm` ⇒ `102cm`
- Exact arithmetic
- Deferred type errors and incomplete results
- Type constraints with abstract supertypes

Cyrus Omar et al. "Toward Semantic Foundations for Program Editors". In: *Summit on Advances in Programming Languagess (SNAPL)*. vol. 71. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017, 11:1–11:12

Andrew John Kennedy. *Programming Languages and Dimensions*. Technical Report 291. 15 JJ Thomson Avenue, Cambridge, United Kingdom: University of Cambridge Computer Laboratory, Apr. 1996

A final digression: array programming

Pioneered in APL,
now in J, K, Python, MATLAB…

Array programming is clearly great for data analysis...
can it be used here too?