EXTENDED ABSTRACT

# Tactile Terms

Philip K.F. Hölzenspies

In this paper, we are arguing for a new way of programmer-compiler interaction. Programming languages with strong type systems have proven their use in disallowing erroneous programs. Arguably, therefore, as many programs as possible should be written in a strongly typed language. Uptake of such languages by the programming community at large, unfortunately, is slow. Worse still, strongly type languages are loosing ground to dynamically typed languages, like Python, Ruby, Perl and PHP. We believe the rejection of strong types by many people stems from the experience that they make for complicated languages. Type errors seem to have an especially deterring effect on students and programming novices. The alternative programmer-compiler interaction discussed in this paper is aimed at making strong types more accessible to the mainstream of programming, as well as providing expert programmers in these languages with a comfortable for education and fast development. This is achieved by giving users per-modification feedback on the types of their terms and a more visual presentation of these terms than simple (syntax highlighted) text.

## 1 Introduction: Error messages

As a motivating introduction to this work, consider compiler error messages. Error messages in strongly typed programming languages are notoriously hard to do well. Good error messages are concise, directly relate to the programmer's code and give an indication of the clash between what was expected and what was found. Error analysis and the production of good error messages is its own field of research [1, 3, 4, 6].

The majority of current research, however, is about making the (textual) message more informative. This leaves a cognitive gap for the user to fill in. We illustrate this with the following example program in Haskell, compiling it with the Glasgow Haskell Compiler:

```
import Control.Applicative
mom, dad :: Person → Maybe Person
granddads p = (,) ⟨$⟩ (mom p >>= dad) ⟨⋆⟩ (dad >>= dad)
```

The compiler informs us we have made a mistake:

```
tfpex.hs:3:44:
    Couldn't match expected type 'Maybe a0'
                with actual type 'Person → Maybe Person'
    In the first argument of '(>>=)', namely 'dad'
    In the second argument of '(⟨⋆⟩)', namely '(dad >>= dad)'
    In the expression: (,) ⟨$⟩ (mom p >>= dad) ⟨⋆⟩ (dad >>= dad)
```

Although this error is very informative, it requires 'translation' (mostly positional) by the programmer, to find the exact place in the program this error pertains to. Consider this as an alternative presentation of the same error:

To appear in EPTCS.

```
import  Control . Applicative
mom,dad::Person → Maybe Person
granddads p = (,) ⟨$⟩ (mom p >>= dad) ⟨⋆⟩ ( dad >>= dad)
                  expression has type  Person → Maybe Person
                  but expecting        Maybe Person
```

This representation presents precisely the same information to the programmer, but without the aforementioned cognitive gap. It is common knowledge among teachers of programming that compiler error messages are often not read very precisely, if at all. There are even experienced programmers who first only look at a line number and try to see whether they see what is wrong before reading the remainder of the error message. In this sense, such more graphical error messages make error information more 'accessible' to the users of a compiler.

This new presentation style also invites us to explore restyling the error information to be even more informative. As a prelude to the remainder of this paper, consider, in this example, the following alternative error presentation:

```
import  Control . Applicative
mom,dad::Person → Maybe Person
granddads p = (,) ⟨$⟩ (mom p >>= dad) ⟨⋆⟩ (dad Person >>= dad)
                                          missing argument
```

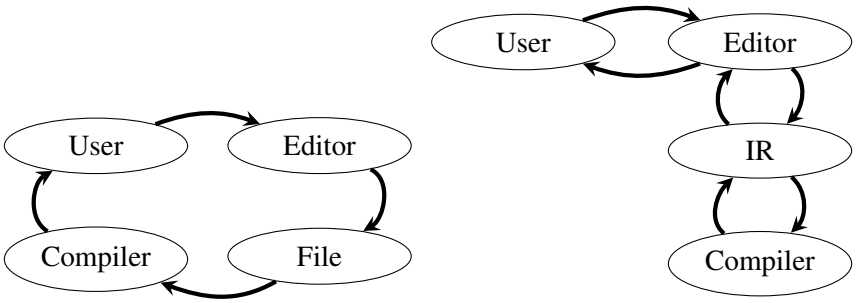
## 2 Related work

Polymorphism considerably worsens the complexity of reporting errors, especially to novice programmers. Work has been done on graphical representations to aide the teaching of polymorphic types [5]. Although this work was based on sample groups that were too small to derive statistically significant conclusions, the results gave promissing indication that type visualisation did assist the understanding of students.

Other work has shown that programming using graphical notations makes programs harder to read [2]. Evidence presented in [7] indicates textual representations are better than visual representations for tasks in general that can be characterised as *symbolic* rather than *spatial*. Programming is itself mostly a symbolic task. The work presented in this paper, therefore, does not substitute textual representations of terms by graphical representations, but rather provides feedback on textual terms in a graphical way. In other words, programming remains a symbolic activity, whereas feedback is presented as graphical annotation of the textually represented program.

## 3 Shifty errors

As noted above, polymorphically typed terms produce even harder to comprehend errors than those that are 'only' strongly typed. Under polymorphism, modifying a program—that passes the scruples of the typechecker—in one place, may produce an error many lines away from that modification. This means that programmers must reverse engineer the type inference to reconstruct where they made an error. In the case where only one change is made to a formerly correct program, somehow informing the typechecker of this change could help localise the error message to this modification. However this requires directing the type inference itself to the corresponding place in the compiler's internal representation of the code.

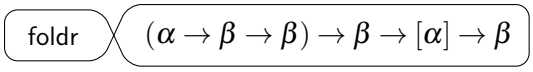



(a) Batch programming (b) Interactive programming

Figure 1: Editor models

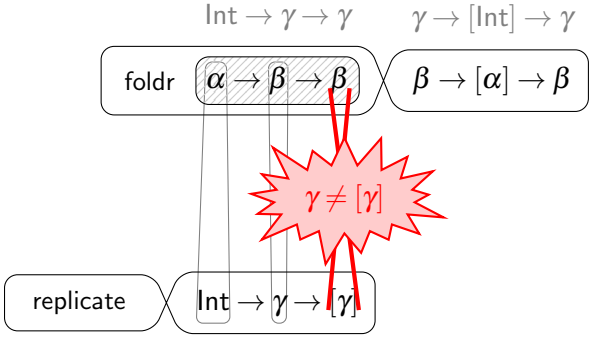
Under current editor models (edit, save, compile, fix errors), this can not generally be done, since there may be multiple changes made in any iteration (see figure 1(a)). Instead, we propose to change the editor model itself and to integrate editing and compilation. As such, the editor becomes a transformer of the compiler's internal representation, as depicted in figure 1(b). This requires a different atomic unit in the editor, i.e. the atomic unit is no longer a character, but rather a term. Although a program is a complete term, the editor's state consists of a set of 'incomplete' terms.

We propose a graphical notation of terms and their types, by example:




This 'term block' can be manipulated and moved. One operation on a term with a function type, as in the example above, is to move an argument from the type to the term. This introduces a 'hole' on the term side of the term block. This whole is represented by a 'blob' with the annotation of the type. As such, this represents a 'shape' of a hole, in terms of type. In the case of this example:


When another term is dragged into a hole, the outer type of the dragged term, must be unifiable with the type in the hole. This term manipulation is a direct manipulation on the internal representation, i.e. it is the substitution of a variable node in the syntax tree by another tree. Without having to parse, this new tree can immediately be type checked. If the type checker rejects the new tree, the term is visually repelled from the hole, with an indication of where the type fails to unify. As an example:



## 4 Focus on the result

As noted by Bret Victor in his seminal presentation at CUSEC2012 [8], a programmer effectively performs the computations she wants her program to perform in her head while writing the program. Since the point of writing a program is to have the computer perform these computations and relieving humans from the burden, so Victor argues, this is an outmoded modus operandi. Instead, the computer should perform the computations on example-input *as the program is being written*.

The same can be said of type-checking. When programming strongly typed languages, the programmer is performing type inference in her head. This poses a classical bootstrapping problem in education; Students have to perform the type inference that they are yet to learn how to do. Having immediate type feedback allows students to focus on the resulting well-typed term, while composing it.

## 5 Status of this work

The ideas discussed in this extended abstract are work in progress. A prototype is currently being developed that will be employed in the education of functional programming in our undergraduate programme. If numbers of students allow, we will be able to have test groups and control groups of sufficient size to draw statistically significant conclusions.

## References

[1] Nabil El Boustani and Jurriaan Hage. Improving type error messages for generic java. In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, PEPM '09, pages 131–140, New York, NY, USA, 2009. ACM.

[2] T.R.G. Green and M. Petre. When visual programs are harder to read than textual programs. In *In*, pages 167–180, 1992.

[3] Jurriaan Hage and Bastiaan Heeren. Heuristics for type error discovery and recovery. In Zoltán Horváth, Viktória Zsók, and Andrew Butterfield, editors, *Implementation and Application of Functional Languages*, volume 4449 of *Lecture Notes in Computer Science*, pages 199–216. Springer Berlin / Heidelberg, 2007. 10.1007/978-3-540-74130-5_12.

[4] Jurriaan Hage and Peter van Keeken. Neon: A library for language usage analysis. In Dragan Gaševic, Ralf Lämmel, and Eric Van Wyk, editors, *Software Language Engineering*, volume 5452 of *Lecture Notes in Computer Science*, pages 35–53. Springer Berlin / Heidelberg, 2009. 10.1007/978-3-642-00434-6_4.

[5] Yang Jun, Greg Michaelson, and Philip W. Trinder. Explaining polymorphic types. *Comput. J.*, 45(4):436–452, 2002.

[6] Vincent Rahli, J. B. Wells, and Fairouz Kamareddine. Technical report HW-MACS-TR-0079: A constraint system for a SML type error slicer.

[7] Iris Vessey. Cognitive fit: A theory-based analysis of the graphs versus tables literature*. *Decision Sciences*, 22(2):219–240, 1991.

[8] Bret Victor. Inventing on principle, 2012. Presentation at the Canadian University Software Engineering Conference (CUSEC) 2012.