Early Experience: Teaching the Basics of Functional Language Design with a Language Type Checker

MATTEO CIMINI, University of Massachusetts Lowell, USA

1 INTRODUCTION

In this short paper, we share our experience in teaching one instance of a course in programming languages. Two key learning outcomes of this course are:

(1) Familiarizing with some basic methods and tools of programming languages theory. In particular, most emphasis has been allocated in defining programming languages the way they are typically shared within the research community: through BNF grammars and inference rule systems for the modeling of type systems and operational semantics, and
(2) Using such tools to the design of toy functional languages that are type sound.

Type soundness is an important feature of modern programming languages. It ensures that the type system at hand faithfully predicts the shapes of values that will be encountered at run-time. In other words, expressions that, at compile-time, are classified of a certain type will actually yield a value of that type at run-time, if evaluated.

The focus of this paper is on outcome (2). One feature that is distinctive of the course is the use of a software tool called *TypeSoundnessCertifier* [Cimini 2015], which we shall describe below.

1.1 TypeSoundnessCertifier

TypeSoundnessCertifier makes use of a domain-specific language to define languages in operational semantics. Below is an example of language definition: a simply typed lambda-calculus with lists.

```
1   Expression E ::= x | (abs T (x)E) | (app E E) | emptyList | (cons E E)
2                    | (head E) | (tail E) | myError
3   Type T ::= (arrow T T) | (list T)
4   Value V ::= (abs T E) | emptyList | (cons V V)
5   Error ::= myError
6   Context C ::= [] | (app C E) | (app V C) | (cons C E) | (cons V C) | (head C) | (tail C)
7
8   Gamma |- (abs T1 E) : (arrow T1 T2) <== Gamma, x : T1 |- E : T2.
9   Gamma |- emptyList : (list T).
10  Gamma |- (cons E1 E2) : (list T) <== Gamma |- E1 : T /\ Gamma |- E2 : (list T).
11  Gamma |- (app E1 E2) : T2 <== Gamma |- E1 : (arrow T1 T2) /\ Gamma |- E2 : T1.
12  Gamma |- (head E) : T <== Gamma |- E : (list T).
13  Gamma |- (tail E) : (list T) <== Gamma |- E : (list T).
14  Gamma |- myError : T.
15
16  (app (abs T E) V) --> E[V/x].
17  (head emptyList) --> myError.
18  (head (cons V1 V2)) --> V1.
19  (tail emptyList) --> myError.
20  (tail (cons V1 V2)) --> V2.
```

This definition describes a grammar, a type system, and an operational semantics, and does so with a syntax that is rather intuitive to those familiar with programming languages theory [Harper 2012; Pierce 2002]. Language definitions are executable through a compilation to lambda-prolog [Miller and Nadathur 2012]. A relevant feature of *TypeSoundnessCertifier* is that the tool **type checks language definitions (which of course are made of syntax), rather than programs**.

Author's address: Matteo Cimini, University of Massachusetts Lowell, USA, Matteo_Cimini@uml.edu.

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

It does so to ensure that the components of the language (value declarations, typing rules, reduction semantics rules, etc.) are all in order so that type soundness automatically holds. Previous approaches to type checking languages are based on intrinsic typing, and achieve soundness indirectly by leveraging on properties of a host meta type theory. The unique feature of *TypeSoundnessCertifier* is that, unlike these other approaches, is based on a type system that explicitly models language design principles and invariants of the (object) language being defined. Three design examples are:

- Classification of the operators of the language in *introduction forms* (such as the abstraction in the simply typed lambda-calculus), *elimination forms* (such as application), *derived operators* (such as `let` and `letrec` in ML-like languages), *errors* and *error handlers* (such as `try` in languages with exceptions).
- If an operational semantics rule requires the argument of an operation to be a value, then that argument must be an evaluation context. For example, this invariant prescribes that the evaluation context $(v\ E)$ be declared when we have the call-by-value function application $(\lambda x.e\ \boxed{v}) \longrightarrow e[\boxed{v}/x]$ (using the meta-variable v requires the argument to be a value). Without such evaluation context, the expression $((\lambda x.x)\ ((\lambda x.x)\ 3))$ would be stuck, jeopardizing type soundness.
- Each elimination form of a type must have a reduction rule defined for *each* of the values of that type. For example, this invariant prescribes that we do not forget to define the behavior of the `head` list operation for empty lists, with a reduction rule (`head emptyList`) $\longrightarrow$ `error`, or otherwise our programs may get stuck, jeopardizing type soundness.

Due to lack of space, we omit discussing the complete list of language design patterns that *TypeSoundnessCertifier* enforces. It is to say that these invariants are certainly not a novelty of the tool. Conversely, they are best practices that language designers have been using for long. The interested reader is invited to refer to our archived paper, which makes these language designs invariants explicit, and formulates them in the context of a formal type system over language definitions [Cimini et al. 2016]. In addition, *TypeSoundnessCertifier* also produces the machine-checked proof of type soundness for those languages that type check successfully, hence the name of the tool. We are not concerned with the automatic certification capabilities of the tool in this short paper. We are, instead, concerned with its ability of pinpointing design mistakes that hinder type soundness. To make a few examples, below are some outputs printed by the tool.

- Suppose that we deleted the rule at line 18 of the code above, *TypeSoundnessCertifier* would print the error message *"Operator **head** is elimination form for the type **list** but does not have a reduction rule for handling one of the values of type **list**: value **emptyList**"*.
- Suppose that we deleted the context declaration (`app C e`) at line 6, *TypeSoundnessCertifier* would print the error message *"The principal argument of the elimination form **app** is not declared as evaluation context, hence some programs may get stuck"*
- Suppose we omit to type check the second argument of `cons` at line 11, *TypeSoundnessCertifier* would print this typing rule saying *"This typing rule does not assign a type to expression **E2**"*
- If we replaced line 19 with (`head (cons E1 E2)`) $\longrightarrow$ `E2`, i.e. now the head operation returns the rest of the list, then *TypeSoundnessCertifier* would print the error message *"Reduction rule of **head** for handling a value **cons** is not type preserving"*.

Thanks to the modeling of language design with a type system, *TypeSoundnessCertifier* has a high-level view on the language being analyzed and can report design mistakes using the same jargon of language designers. Because of this, we believe that *TypeSoundnessCertifier* can be an effective tool for teaching the basics of functional language design to those students that are engaging in a programming languages course and are exposed to programming languages theory, including its jargon. Furthermore, the tool can be used to collect statistics on the frequency of

design mistakes that students may make during their language design exercises. These statistics may guide best practices in teaching language design.

The title of this paper points out that we focus on functional languages only. With this we mean pure languages in which the computation is a rewrite of an expression into another expression. In more technical words, we have restricted ourselves to an operational semantics relation with shape $e \longrightarrow e$. This is opposed to languages that, for example, also affect a memory during computaton.

The title also points out that, within this class of languages, only basic designs have been addressed. In particular, we have restricted ourselves to language design patterns of the like of those that have been pointed out above. These are certainly not enough to guarantee type soundness for modern functional languages features such as modules, monads, type classes, macros, and other complex features. They are enough, however, to define simple functional languages.

Furthermore, the title stresses that this is an early experience. This short paper is based, indeed, on only one instance of a course. Furthermore, the course had low enrollment.

In this paper, we give details about the course for the benefit of instructors that would like to replicate/adapt this teaching experience. We also share some simple statistics and our impressions about them, but we make sure that it be clear that no conclusion can be drawn from our experience. More, and more extended experiments should be conducted. Next section offers details on the course, some data from our evaluations and our impressions.

2 THE COURSE IN SOME DETAIL

The name of the course is *Design of Programming Languages*, and has been a semester long (Spring 2018) graduate level course. The course has been divided into three parts: Programming languages theory, advanced features (but no formal theory), and gradual typing. Only the first part is relevant to this paper, and we shall give some details as to what it comprised: in class, outside of the class, and during the evaluation. In replicating/adapting this teaching experience, other choices are certainly worthy. For example, courses that are similar to ours often expand the part on programming languages theory to cover advanced features with mathematical rigor.

In Class. The course has spanned 15 weeks and the part on programming languages theory has been covered in the first half of the course (7 weeks, Midterm included). The textbook that has been adopted for this part is TAPL [Pierce 2002], of which we have covered the typical chapters on the simply typed lambda-calculi with common datatypes, and the chapters on languages with references, and subtyping. One week has been devoted to interactive theorem proving and tools for language definitions. In this context, the instructor demonstrated with substantial depth the *TypeSoundnessCertifier* tool over examples from TAPL.

Outside of the Class. Students were informed that a part of the evaluation was to show language modeling skills in front of the instructor with the use of *TypeSoundnessCertifier*. Students have been encouraged to download the tool and practice using it outside of the class. *TypeSoundnessCertifier* also has a repository of examples that includes a number of language definitions that are automatically checked as type sound. Students have been invited to browse and reason over these examples. To encourage practice, the instructor invited the students to model the following features with *TypeSoundnessCertifier*. (None were mandatory).

- `reverseRange` e, as in: `reverseRange` $3 \longrightarrow^* [3,2,1]$ (A reversed range is somehow easier).
- `length` e, as in: `length` $[4,5,7] \longrightarrow^* 3$
- `map`, the typical operator in functional programming languages.

The Evaluation. Each student arranged an appointment to meet with the instructor at the instructor's office. Students have been evaluated in the context of this meeting. This appointment

was individual for each student. The student could use her own laptop with *TypeSoundnessCertifier* installed, or use the instructor's laptop. (Other options, such a using lab's computers were possible but did not occur.) The instructor assigned the student with the task of modeling a simple language with *TypeSoundnessCertifier*. All students have been assigned the same language, which we here call **filterOpt**, and whose details are given below. The student did know that this part of the evaluation would consist of modeling a language but did not know what specific language would be assigned prior to the meeting. The student had 20 minutes to complete the task. The task was completed successfully when **filterOpt** was modeled and *TypeSoundnessCertifier* said that it was a type sound language definition. During language modeling, the student could invoke *TypeSoundnessCertifier* as many times as she pleased. The number of failed attempts and the nature of the mistakes reported by the tool did not affect her grade. The grade was assigned based on how close the language definition of the student was to the requested solution after 20 minutes. Roughly speaking, this time limit did not count the time that was not spent in using modeling skills or interacting with the tool to realize such skills. To make an example, if the student did not remember that inference rules must end with '.' in the syntax of *TypeSoundnessCertifier*, she may have entertained with the parser, or possibly asked the instructor. This time did not count. The rational for this is that the focus of the instructor was exclusively on evaluating the language modeling skills of the student, not her memorization skills w.r.t. the domain-specific language at hand. In replicating/adapting this teaching experience, another instructor may have a different take on this aspect.

*The **filterOpt** (Toy) Language.* For this evaluation, the student was provided with an existing language definition that included functions, booleans, if-then-else and lists. The task that has been asked prescribed to extend this language definition with

- option types (with operators `none`, `some` e, `get` e), and
- an operator called `filterOpt`, a variant of the filter operation in functional programming. `filterOpt` l f takes a list l of elements of type T, and a function f from T to booleans, and creates a list in which every element v of l is (`some` v), if $f(v)$ is true, or `none`, if $f(v)$ is false.

To model a type sound language, the student was called to update the syntax for expressions, values, types, and evaluation contexts, as well as adding appropriate typing rules and reduction semantics rules. In *TypeSoundnessCertifier*, a possible solution could be the following. (Below, we show only the relevant part and omit the rest using dots . . .)

```
1  Expression E ::= ... | none | (some E) | (get E) | (filterOpt E E)
2  Type T ::= ... | (option T)
3  Value V ::= none | (some V)
4  Error ::= myerror
5  Context C ::= ... |  (some C) | (get C) | (filterOpt C E) | (filterOpt V C)
6
7  Gamma |- none : (option T).
8  Gamma |- (some E) : (option T) <== Gamma |- E : T.
9  Gamma |- (get E) : T <== Gamma |- E : (option T).
10 (get none) --> myerror.
11 (get (some V)) --> V.
12
13 Gamma |- (filterOpt E1 E2) : (list (option T)) <== Gamma |- E1 : (list T)
14                                                    /\ Gamma |- E2 : (arrow T bool).
15 (filterOpt emptyList V) -->  emptyList.
16 (filterOpt (cons V1 V2) V3) --> (cons (if (app V3 V1) (some V1) none) (filterOpt V2 V3)).
```

2.1 Report on our Experience[1]

The course had 11 students. We have kept note of the mistakes that *TypeSoundnessCertifier* reported during evaluations. Below, we share some data about these evaluations, and our impressions about

[1]The instructor is authorized to report this data on the ground of an approved IRB and informed consents.

them. It is important to notice that the total number of evaluations (11) is largely insufficient, and we cannot draw general conclusions from our data.

We show only some of the statistics that stand out:

- 5 out of 11 students forgot to declare some evaluation contexts. Our impression is that, for future experiments, this aspect should be monitored: If, in future, further data will support a conclusion that this mistake could be frequent, it may be worthy to adjust the teaching style in class towards allocating much more emphasis to evaluation contexts.
- 4 out of 11 students failed in defining a type preserving reduction rule. Again, we should monitor if future experiments may support a conclusion that this could be a frequent mistake. If so, this mistake may be very much similar to the encountering of type checking errors in programming. Overall, we are hardly worried if our students do not write a perfectly well-typed program at first attempt. Therefore, whether or not this mistake turns out to be frequent, it may not call for particular adjustments in teaching.
- 6 out of 11 students completed the task successfully. One student succeeded at the 6th attempt, one succeeded at the 4th attempt, three students succeeded at the 3rd attempt. In these cases, students received a feedback from *TypeSoundnessCertifier* in the form of an error message and returned to the language definition to fix it. One student succeeded at first attempt. On average, students who completed the task invoked the tool 2.333 times before succeeding.

2.2 Conclusions

Our work strives to demonstrate that a language type checker can be an effective tool in teaching language design. We have used *TypeSoundnessCertifier* in the context of an instance of a course in programming languages. We have offered details on how the course took place, and reported on some data gathered during our evaluations. Although the work here reported is not statistically significant, we hope to inspire our colleagues to adopt a language type checker such as *TypeSoundnessCertifier* in their courses in programming languages. The fact that 6 out of 11 students could complete a language design task through the feedback of the tool is encouraging in the context of our small experience. It was certainly a joy to see students try their language definitions against the tool, receive feedback, fix accordingly, and ultimately succeed.

In future, we would like to replicate the course experience several times, and report on our extended findings to the community. We especially would like to conduct large studies that collect statistics on the frequency of language design mistakes. These statistics may be precious to inform best practices in teaching language design.

We would like to expand the course to make subtyping a relevant part of the evaluation. (*TypeSoundnessCertifier* handles subtyping but this feature was not part of the language **filterOpt**). We also would like to extend *TypeSoundnessCertifier* to languages with stores, and experiment on teaching the most common designs insofar this class of languages is concerned. Finally, we would like to empower *TypeSoundnessCertifier* with automatic grading capabilities in the style of Automata Tutor [D'Antoni et al. 2015].

REFERENCES

Matteo Cimini. 2015. *TypeSoundnessCertifier*. https://github.com/mcimini/TypeSoundnessCertifier.

Matteo Cimini, Dale Miller, and Jeremy G. Siek. 2016. Well-Typed Languages are Sound. *CoRR* abs/1611.05105 (2016). arXiv:1611.05105 http://arxiv.org/abs/1611.05105

Loris D'Antoni, Matthew Weavery, Alexander Weinert, and Rajeev Alur. 2015. Automata Tutor and what we learned from building an online teaching tool. *Bulletin of the EATCS* 117 (2015).

Robert Harper. 2012. *Practical Foundations for Programming Languages.* Cambridge University Press.

Dale Miller and Gopalan Nadathur. 2012. *Programming with Higher-Order Logic.* Cambridge University Press.

Benjamin C. Pierce. 2002. *Types and Programming Languages.* MIT Press.