Language Transformations in the Classroom

Matteo Cimini
University of Massachusetts Lowell
Lowell, Massachusetts, USA
matteo_cimini@uml.edu

Benjamin Mourad
University of Massachusetts Lowell
Lowell, Massachusetts, USA
benjamin_mourad@student.uml.edu

Language transformations are algorithms that take a language specification in input, and return the language specification modified. Language transformations are useful for automatically adding features such as subtyping to programming languages (PLs), and for automatically deriving abstract machines.

In this paper, we set forth the thesis that teaching programming languages features with the help of language transformations, in addition to the planned material, can be beneficial for students to help them deepen their understanding of the features being taught.

We have conducted a study on integrating language transformations into an undergraduate PL course. We describe our study, the material that we have taught, and the exam submitted to students, and we present the results from this study. Although we refrain from drawing general conclusions on the effectiveness of language transformations, this paper offers encouraging data. We also offer this paper to inspire similar studies.

1 Introduction

Computer Science university curricula include undergraduate courses in programming languages (PLs). These courses vary greatly in the content they offer, and they may also have various names such as "Principles of Programming Languages", and "Organization of Programming Languages", to make some examples. Typically, the goal of these courses is not to teach one specific PL. Conversely, students are exposed to the conceptual building blocks from which languages are assembled, the various programming paradigms that exist, and students are challenged to think about various PL features in their generality.

It is typical for these courses to cover PL features such as subtyping, abstract machines, type inference, parametric polymorphism, as well as many others. Some of these features can be regarded as variations on a base PL. For example, it is not uncommon to design a PL, and add subtyping *afterwards*. It is then interesting to understand what are the modifications that need to take place in order to incorporate subtyping in that base language. A good way to analyze this is by looking at how formal typing rules need to change. Consider, for example, the typing rule of function application below on the left, and its version with (algorithmic) subtyping on the right.

$$\text{(T-APP)}\ \frac{\Gamma \vdash e_1 : T_1 \to T_2 \quad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\, e_2 : T_2} \quad \Longrightarrow \quad \text{(T-APP')}\ \frac{\Gamma \vdash e_1 : T_{11} \to T_2 \quad \Gamma \vdash e_2 : T_{12} \quad T_{12} <: T_{11}}{\Gamma \vdash e_1\, e_2 : T_2}$$

(T-APP) rejects programs that pass an integer to a function that works on floating points, such as the program $((\lambda x : \texttt{float}.x)\ 3)$, where $\emptyset \vdash 3 : \texttt{int}$. This is because the type T_1 in $T_1 \to T_2$, which is the domain of the function e_1, needs to be the exact same type T_1 of the argument e_2. If we were to add subtyping, such a parameter passing would be accepted by the type system.

The first modification that (T-APP') makes of (T-APP) is to let the domain of the function and the argument have different types. To do so, (T-APP') assigns two different variables to the two occurrences of T_1, that is, T_{11} for the domain of the function, and T_{12} for the type of the argument. Next, (T-APP') needs to understand how T_{11} and T_{12} are related by subtyping. As T_{11} appears in contravariant position in $T_{11} \to T_2$, it means that T_{11} describes the type of an input. The argument e_2 will be provided as a value. Therefore, it is the type T_{12} of the argument that must be a subtype of T_{11}, rather than the other way around, for example. Hence, the subtyping premise $T_{12} <: T_{11}$ is added to (T-APP').

We can describe these modifications with an algorithm that takes (T-APP), and automatically transforms it into (T-APP'). To summarize, such an algorithm must perform two steps:

- **Step 1**: Split equal types into fresh, distinct variables, and
- **Step 2**: Relate these new variables by subtyping according to the variance of types.

This type of algorithm can be formulated over a formal data type for language specifications. In other words, we can devise a procedure that takes a language specification in input (as a data type), and returns another language specification (with subtyping added). These algorithms are called *language transformations* [18]. One of the benefits of language transformations is that they do not apply just to one language. Instead, they can apply to several languages. For example, the two steps above can add subtyping for types other than the function type, such as pairs, lists, option types, and other simple types.

Our Thesis Another benefit of language transformations is that they highlight the central insights behind a feature being added. For example, **Step 1** and **Step 2** are key aspects of subtyping. Teaching students the algorithms that automatically apply **Step 1** and **Step 2** to languages can provide them with a firmer grasp of the concept of subtyping overall.

The approach is not limited to subtyping. The adding of other PL features can be formulated as language transformations, and taught to students in class as well. We think that exposing students to the language transformations for adding PL features may constitute a good addition in the classroom.

In this regard, however, we point out that we do not advocate for teaching PL features exclusively with the sole help of language transformations. For example, we teach subtyping using the material in the TAPL textbook [19], and we are skeptical that it would be a good idea to skip this material before introducing language transformations. This is because language transformations constitute quite a technical deep dive, and students could benefit from a gentler introduction of PL concepts.

Ultimately, in this paper we set forth the thesis that *using language transformations for teaching PL features, in addition to the planned material, can be beneficial for students to deepen their understanding of the features being taught.*

Contributions of this Paper We have experimented with teaching the language transformations for adding subtyping and deriving CK abstract machines [14]. We have conducted our study on two instances of an undergraduate course on programming languages.

In class, we first have introduced subtyping with material from TAPL [19], as mentioned above, and then we have taught the language transformations for adding subtyping (which we describe in Section 2.1). To evaluate whether our students gained a good understanding of subtyping, the final exam presented them with a language with operators that are not standard. Then, the exam asked students to add subtyping to these operators based on the language transformations that they have learned.

In the context of this study, we have collected information about students' success in providing a correct answer to such a task. We describe the final exam in detail in Section 2.3, and we report on the results of this study in Section 3.

We have taught the topic of CK machines following the notes of Felleisen and Flatt [14]. We then have taught the language transformations for deriving CK machines (which we describe in Section 2.2). Analogously to subtyping, the final exam asked our students to derive the CK machine for a language with operators that are not standard. We then have collected information about students' exam answers for this task, and we report on this data in Section 3. In total, the study involved 55 undergraduate students.

To summarize our contributions, in this paper:

- We set forth the thesis that language transformations can be a beneficial addition in PL courses, as formulated above.
- We describe the study that we have conducted, which includes the material that we have taught, and the exam submitted to the students.
- We present the results from our study. Although we explicitly say that we should not consider our results conclusive, the data that we present is encouraging.
- We offer this paper to inspire similar studies towards gathering evidence for, or against, our thesis.

Roadmap of the Paper Section 2 describes the study that we have conducted, Section 3 presents our results, Section 4 describes our future work, and Section 5 concludes the paper.

2 Language Transformations in Class

General Details about the Course The course is at the undergraduate level, and is based on the TAPL textbook [19]. The course covers the typical topics of PL theory on defining syntax (BNF grammars), operational semantics, and type systems of PLs. The course also covers several other topics such as parameter passing, scoping mechanisms, subtyping, abstract machines, recursion, exceptions, dynamic typing, memory management, concurrency, and logic programming. Students are then familiar with the formalisms of operational semantics and type systems when the course covers the topics of subtyping and abstract machines.

The evaluations of the course include a long-term programming project in which students develop an interpreter for a functional language with references in OCaml, and a final exam with questions and open answers at the end of the course. The final exam tests our students on the topics of subtyping and abstract machines. We will describe our exam in Section 2.3.

Algorithms in Pseudo-Code Language transformations are algorithms, which begs the question on what syntax we should use to describe them. We took inspiration from courses in Algorithms and Data Structures, and from textbooks such as [10], where algorithms are described in pseudo-code. Therefore, we have used a pseudo-code that, to our estimation, was always intuitive to students, even though we did not thoroughly and precisely define it (as in [10]).

Language Specifications During the course, students acquire familiarity with formal definitions of programming languages, which they learn through TAPL. To recap, languages are defined with a BNF grammar and a set of inference rules. Inference rules are used to define a type system, a reduction relation, and auxiliary relations, if any. To make an example, we repeat the typical formulation of the simply-typed λ-calculus. We use a small-step operational semantics and evaluation contexts.
(Below, B is some base type.)

$$\begin{array}{llcl}
\text{Type} & T & ::= & T \to T \mid B \\
\text{Expression} & e & ::= & x \mid \lambda x{:}T.\, e \mid e\; e \\
\text{Value} & v & ::= & \lambda x{:}T.\, e \\
\text{Evaluation Context} & E & ::= & [\cdot] \mid E\; e \mid v\; E \\
\text{Type Environment} & \Gamma & ::= & \emptyset \mid \Gamma, x : T
\end{array}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash (\lambda x{:}T_1.\, e) : T_1 \to T_2} \qquad \overset{\text{(T-APP)}}{\frac{\Gamma \vdash e_1 : T_{11} \to T_{12} \qquad \Gamma \vdash e_2 : T_{11}}{\Gamma \vdash e_1\; e_2 : T_{12}}}$$

$$((\lambda x{:}T.e)\; v) \longrightarrow e[v/x] \qquad \frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}$$

We allow our pseudo-code to refer to parts of a language specification. For example, if the language L is the formulation of the simply-typed λ-calculus, then it contains both the grammar and the set of inference rules above. *L.rules* retrieves the set of inference rules. Given a rule r, say (T-APP), *r.premises* retrieves the set of premises of (T-APP), which are the formulae above the horizontal line. *r.conclusion* retrieves the formula below the horizontal line. To our estimation, these references in the pseudo-code, as well as all other references, are rather intuitive, and they will be clear when we use them. (This is also the take on pseudo-code that [10] has, where a number of operations are not defined beforehand.)

Roadmap of this Section Below, we describe our experiment on teaching subtyping (Section 2.1) and CK machines (Section 2.2). We also describe the final exam given to students (Section 2.3). Our pseudo-code for adding subtyping is based on an algorithm expressed in a domain-specific language in [18]. We are not aware of any analogous algorithm that corresponds to our pseudo-code for deriving CK machines. The next sections describe the algorithms that we have taught in class, of which we do not claim any theoretical results of correctness.

2.1 Language Transformation for Subtyping

In class, we have taught subtyping based on the corresponding chapters in the TAPL textbook [19]. Then, we have taught language transformation algorithms for adding subtyping to simple functional languages. The task of adding subtyping has been divided into the two steps that we have discussed in the introduction: 1) Split equal types, and 2) Relate new variables by subtyping according to the variance of types.

Split Equal Types This step modifies the typing rules of a language so that the variables that occur more than once in their premises are given different variable names. As we have seen in the introduction, this is the first step to let different expressions have different types. We define the procedure

SPLIT-EQUAL-TYPES to perform this step. The pseudo-code of SPLIT-EQUAL-TYPES is given below, which we explain subsequently.

```
SPLIT-EQUAL-TYPES$(P)$
1  $newPremises = \emptyset, varmap = \emptyset$
2  for each $p \in P$
3       if $p$ is of the form $\Gamma \vdash e : someType$
4            for each $T \in someType$ s.t. $T$ appears more than once in $P$
5                 $T' = $ FRESH$(P)$
6                 $p = p$ where $T'$ replaces $T$ in $someType$
7                 $varmap(T) = varmap(T) \cup \{T'\}$
8       $newPremises = newPremises \cup \{p\}$
9  return $(newPremises, varmap)$
```

SPLIT-EQUAL-TYPES takes a set of premises P in input, and returns a pair with two components: a set of premises *newPremises*, and a map *varmap*. Here, *newPremises* is the same set of premises P in which each variable has been given fresh, distinct names, if occurring multiple times. *varmap* maps each of the variables that have been replaced to the set of new variables that replaced them. The reason for collecting these new variables in *varmap* is because we need to relate them by subtyping. (This is the responsibility of the second step, which works based on the information in *varmap*.)

To make an example, when SPLIT-EQUAL-TYPES is applied to the premises of (T-APP), we have
Input: $P = \{\Gamma \vdash e_1 : T_1 \to T_2, \Gamma \vdash e_2 : T_1\}$
Output = $(newPremises, varmap)$ where

$$\begin{aligned} newPremises &= \{\Gamma \vdash e_1 : T_{11} \to T_2, \Gamma \vdash e_2 : T_{12}\} \\ varmap &= \{T_1 \mapsto \{T_{11}, T_{12}\}\} \end{aligned}$$

SPLIT-EQUAL-TYPES produces this output in the following way. Line 1 initializes *newPremises* to the empty set, and *varmap* to the empty map. The loop at lines 2-8 is executed for each premise p of the set of premises P. For example, with (T-APP) we have two iterations; the first is with $p = \Gamma \vdash e_1 : T_1 \to T_2$, and the second is with $p = \Gamma \vdash e_2 : T_1$. Line 3 extracts the components of the typing premise. It does so in a style that is reminiscent of pattern-matching. The component that is relevant for the algorithm is *someType*, which is the output type of the typing premise. The loop at lines 4-7 applies to each variable T in *someType* that appears more than once in the premises of P. We focus on variables that have multiple occurrences because variables that occur only once do not need to be replaced with new names. For each of these variables T, we generate a fresh variable that is not used in P. We do so with FRESH(P) at line 5. Line 6 modifies the premise p by rewriting it to use the fresh variable in lieu of T. Line 7 also updates *varmap* to add the fresh variable to the set of new variables mapped by T. Line 8 adds p to *newPremises*. At that point, p may have been modified with line 6, or may have remained unchanged. Finally, line 9 returns the pair $(newPremises, varmap)$.

Relate New Variables by Subtyping This second step is performed in the context of the procedure ADD-SUBTYPING. This is our general procedure that takes a language specification in input, and adds subtyping to it. To do so, ADD-SUBTYPING calls SPLIT-EQUAL-TYPES, and then works on the rules modified by SPLIT-EQUAL-TYPES to relate the new variables by subtyping.

The pseudo-code of ADD-SUBTYPING is the following.

```
ADD-SUBTYPING($L$)
1  for each rule $r \in L.rules$ s.t. $r.conclusion$ is of the form $\Gamma \vdash e : someType$
2      $(newPremises, varmap) =$ SPLIT-EQUAL-TYPES($r.premises$)
3      for each mapping $(T \mapsto setOfNewVars)$ in $varmap$
4          if there exists a type in $setOfNewVars$ that is invariant in $newPremises$
5              for each $T_1, T_2 \in setOfNewVars$
6                  $newPremises = newPremises \cup \{T_1 = T_2\}$
7          elseif there is exactly one type $T'$ in $setOfNewVars$ that is contravariant in $newPremises$
8              for each $T_{new} \in (setOfNewVars \setminus T')$
9                  $newPremises = newPremises \cup \{T_{new} <: T'\}$
10         elseif none in $setOfNewVars$ is contravariant or invariant in $newPremises$
11             say that $setOfNewVars = \{T_1, T_2, \ldots, T_n\}$
12             $newPremises = newPremises \cup \{T = T_1 \vee T_2 \vee \ldots \vee T_n\}$
13         else error
14     $r.premises = newPremises$
```

The argument L is the language specification in input. The procedure modifies the rules of L in-place. Line 1 selects only the typing rules of L (leaving out reduction rules, for example). It does so by selecting only the rules whose conclusion has the form of a typing formula. Lines 2-14 constitute the body of the loop, and apply for each of these rules. Line 2 calls SPLIT-EQUAL-TYPES, passing the premises of the typing rule as argument. This call returns the new premises and the map previously described. Lines 3-13 iterate over the key-value pairs of the map. Key-value pairs are of the form $T \mapsto setOfNewVars$, where T is the variable that occurred in the original typing rule before calling SPLIT-EQUAL-TYPES. We dub T as the *original variable*. Also, *setOfNewVars* contains the new variables generated by SPLIT-EQUAL-TYPES for T.

Lines 4-6 cover the case for when the original variable appeared in invariant position. In that case, there exists a variable in *setOfNewVars* that is in invariant position in *newPremises*, which we check with line 4. As the original variable appeared in invariant position, all the new variables must be related by equality. (We make an example shortly). Therefore, lines 5-6 add an equality premise for every two variables in *setOfNewVars*. This case covers operators such as the assignment in a language with references, as T is invariant in a reference type `Ref` T. Consider the typing rule for the assignment operator on the left, and its version with subtyping on the right.

$$\text{(T-ASSIGN)}\ \frac{\Gamma \vdash e_1 : \texttt{Ref}\ T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 := e_2 : \texttt{unitType}} \quad \Longrightarrow \quad \text{(T-ASSIGN')}\ \frac{\Gamma \vdash e_1 : \texttt{Ref}\ T_1 \qquad \Gamma \vdash e_2 : T_2 \qquad T_1 = T_2}{\Gamma \vdash e_1 := e_2 : \texttt{unitType}}$$

Here, SPLIT-EQUAL-TYPES replaces T with two new variables T_1 and T_2, but as T is invariant in (T-ASSIGN), we generate the premise $T_1 = T_2$, which is the correct outcome.

Lines 7-9 cover the case for when the original variable appeared in a contravariant position. In that case, there exists a type T' in *setOfNewVars* that is contravariant in *newPremises*. We detect such a case with line 7. Notice that line 7 also checks that the original variable appeared only once in contravariant position. We address this aspect later when we discuss line 13. As T' appears in contravariant position, this is an input that is waiting to receive values. Therefore, we generate the subtyping premises that set all the other new variables in *setOfNewVars* as subtypes of T' (lines 8-9). This case covers operators such as the function application, which we have discussed previously. Thanks to lines 7-9, ADD-SUBTYPING generates the typing rule (T-APP') from (T-APP), which is the correct outcome.

Lines 10-12 cover the case in which variance does not play a role. In this case, all the newly generated variables are peers. (We will make an example shortly). Therefore, we compute the join $\vee$ for them [19]. This case applies to operators such as if-then-else. Consider the typing rule of if-then-else below on the left, and its version with subtyping on the right.

$$\text{(T-IF)}\ \frac{\Gamma \vdash e_1 : \texttt{Bool} \qquad \Gamma \vdash e_2 : T \qquad \Gamma \vdash e_3 : T}{\Gamma \vdash (\mathit{if}\, e_1\, e_2\, e_3) : T} \quad \Longrightarrow \quad \text{(T-IF')}\ \frac{\Gamma \vdash e_1 : \texttt{Bool} \qquad \Gamma \vdash e_2 : T_1 \qquad \Gamma \vdash e_3 : T_2 \qquad T = T_1 \vee T_2}{\Gamma \vdash (\mathit{if}\, e_1\, e_2\, e_3) : T}$$

Here, SPLIT-EQUAL-TYPES replaces T with two new variables T_1 and T_2. Then, line 10 detects that variance does not play a role for these new variables. Indeed, the two branches of the if-then-else are peers. Therefore, lines 11 and 12 generate the premise that computes the join of all the new variables, and assign it to T. Thanks to lines 10-12, (T-IF') is precisely the typing rule that ADD-SUBTYPING generates, which is the correct outcome. Another example where variables are peers is with the `case` operator of the sum type.

Line 13 throws an error if none of the previous cases apply. This happens, for example, if a variable appears in contravariant position multiple times in the typing rule. Consider the following typing rule.

$$\frac{\Gamma \vdash e_1 : T \to (T \to \texttt{Bool}) \qquad \Gamma \vdash e_2 : T \times T}{\Gamma \vdash \texttt{app2}\, e_1\, e_2 : \texttt{Bool}}$$

Here, T appears in contravariant position twice in the type of e_1. However, the typing rule of app2 cannot distinguish how the components of the pair e_2 are going to be used. Consider two alternative reduction rules for app2:

$$\texttt{app2}\, e_1\, e_2 \longrightarrow ((e_1\ (\texttt{fst}\, e_2))\ (\texttt{snd}\, e_2)) \qquad \mathit{or} \qquad \texttt{app2}\, e_1\, e_2 \longrightarrow ((e_1\ (\texttt{snd}\, e_2))\ (\texttt{fst}\, e_2))$$

The reduction rule on the left entails that the first component of the pair e_2 must be subtype of the first T of $T \to (T \to \texttt{Bool})$, and that the second component of the pair e_2 must be subtype of the second T of $T \to (T \to \texttt{Bool})$. Conversely, the reduction rule on the right entails that the second component of the pair e_2 must be subtype of the first T of $T \to (T \to \texttt{Bool})$, and that the first component of the pair e_2 must be subtype of the second T of $T \to (T \to \texttt{Bool})$.

However, ADD-SUBTYPING only analyzes the typing rule of app2, which alone is not informative enough to tell about the parameter passing to e_1. Therefore, we do not know what subtyping premises to generate. In this case, ADD-SUBTYPING throws an error.

To solve this problem, we could extend ADD-SUBTYPING to analyze the reduction semantics of app2, but we observe that language designers may specify such semantics in a way that is as complex as they wish. Reduction rules may not use parameter passing immediately and evidently, in favor of jumping from operator to operator several times, which makes the analysis hard to do. For these reasons, we have not investigated this path, also because we may be speaking about cases that are quite uncommon, and not strictly necessary to cover in detail in our undergraduate class.

Finally, line 14 replaces the premises of r with *newPremises*. The relations $\vee$ and $<:$ can be generated with an algorithm, too, but we omit showing these procedures here. In this paper, we simply want to illustrate the approach rather than strive for completeness.

2.2 Language Transformation for CK

We have taught abstract machines following the notes of Felleisen and Flatt [14]. To recap, the CK machine remedies an inefficiency aspect of the reduction semantics. Consider the following reductions: (hd retrieves the head of a list, t and f are the constants for the true and false boolean, respectively).

$$(\texttt{if}\ (\texttt{hd}\ [\texttt{f} \wedge \boxed{((\lambda x.x)\ \texttt{t})}, \texttt{t}])\ e_1\ e_2) \longrightarrow (\texttt{if}\ (\texttt{hd}\ [\texttt{f} \wedge \texttt{t}, \texttt{t}])\ e_1\ e_2)$$

$$(\texttt{if}\ (\texttt{hd}\ [\boxed{\texttt{f} \wedge \texttt{t}}, \texttt{t}])\ e_1\ e_2) \longrightarrow \ldots$$

To perform the step at the top, the reduction semantics traverses the term and seeks for the first available evaluation context, which points to the highlighted subterm. At the second step, the reduction semantics must seek again for an available evaluation context, and does so by traversing the term again from the top level if operator, which is inefficient.

To improve on this aspect, and avoid these recomputations, the CK machine carries a *continuation* data structure at run-time. The grammar for continuations, and the CK reduction rules for function application are the following.
(mt is the empty continuation, which denotes machine termination.)

$$\text{Continuation}\ k \quad ::= \quad \texttt{mt} \mid (\texttt{app}_1\ e\ k) \mid (\texttt{app}_2\ v\ k)$$

$$\begin{array}{rll}
(\texttt{app}\ e\ e_2), k \longrightarrow e, (\texttt{app}_1\ e_2\ k) & & \texttt{Start} \\
v, (\texttt{app}_1\ e\ k) \longrightarrow e, (\texttt{app}_2\ v\ k) & & \texttt{Order} \\
v, (\texttt{app}_2\ (\lambda x.e)\ k) \longrightarrow e[v/x], k & & \texttt{Computation}
\end{array}$$

The reduction relation has the form $e,k \longrightarrow e,k$, where k is built with continuation operators mt, $\texttt{app}_1$, and $\texttt{app}_2$. There is a continuation operator for each evaluation context. Each continuation operator has always an argument k, which is the next continuation, and one expression argument less than the operator because one of the expressions is currently out to be the focus of the evaluation. For example, $(\texttt{app}_2\ v\ k)$ means that the current expression being evaluated returns as the second argument of the application, and v is the function waiting for such argument.

Below, we show the language transformations for generating the CK machine, except for the procedure that generates the Computation rule above, because that procedure is straightforward.

Generating the Grammar for Continuations The following pseudo-code generates the grammar Continuation.

```
CK-GENERATE-GRAMMAR($EvalCtx$)
1  create grammar category Continuation, and add grammar item mt to it
2  for each $(\texttt{op}\ t_1 \ldots t_n) \in EvalCtx$
3      if $t_i = E$
4          add grammar item $(\texttt{op}_i\ (t_1 \ldots t_n$ minus $E)\ k)$ to Continuation
```

For each evaluation context, the index where the E appears determines the index of the continuation operator. The arguments of this operator are all the arguments that are not E. Indeed, the argument at that position will currently be the focus of the evaluation. Also, the next continuation k is the last argument.

Generating the `Start` rule The following pseudo-code generates the reduction rule `Start`, which brings the computation of an operator into using continuation operators.

```
CK-GENERATE-START($Continuations$)
1  find $(\texttt{op}_i\, t_1 \ldots t_n\, k) \in Continuations$ with no $v$
2  add reduction rule $(\texttt{op}\, t_1 \ldots\ e\ \ldots t_n), k \longrightarrow e, (\texttt{op}_i\, t_1 \ldots t_n\, k)$
```

Here, e appears at position i in `op`. If a continuation contains some v as arguments, it means that those arguments must have been the subject of some other evaluation context that evaluated them to a value. Therefore, that cannot be the starting point. Our starting point, instead, is a continuation that contains no v. The reduction rule that we add takes the operator into using the continuation operator that we have just found.

Generating `Order` rules The following pseudo-code generates the reduction rules `Order`. These rules evaluate the arguments of the operator by jumping from one continuation operator to another in the order established by the evaluation contexts.

```
CK-GENERATE-ORDER($Continuations, EvalCtx$)
1  for each $(\texttt{op}_i\, t_1 \ldots t_m\, k) \in Continuations$
2      find $(\texttt{op}\, t'_1 \ldots t'_n) \in EvalCtx$ where $(t_k = t'_k$ or $t'_k = E,$ for all $k)$
3      if $t'_j = E$
4          add reduction rule $v, (\texttt{op}_i\, t_1 \ldots t_m\, k) \longrightarrow t_j, (\texttt{op}_j\, t_1 \ldots\ v\ \ldots t_m\, k)$
```

Here, v appears at position i in $\texttt{op}_j$. After finishing an evaluation in the contexts of the continuation $\texttt{op}_i$, we need to find the next continuation operator $\texttt{op}_j$. To do so, we find a match between the arguments of the continuation $\texttt{op}_i$ with arguments of an evaluation context. This is because arguments that are values in the continuation then need to be values, too, in the next evaluation context. Arguments that are simply expressions e in the continuation then need to be expressions e, too, in the next evaluation context. The evaluation context will have, however, an argument E (and only one argument E) at some position j, which identifies the index of the next continuation operator. The reduction rule that we add starts from a point where a value has been computed, and we are in the context of the continuation $\texttt{op}_i$. In one step, we extract the j-th argument of the continuation $\texttt{op}_i$ because that is the expression that now needs to be in the focus of the evaluator. The next continuation is then $\texttt{op}_j$, where we placed the value v just computed among the arguments of $\texttt{op}_j$, and specifically at position i.

Generating `Computation` rules is rather straightforward, hence we omit showing that simple procedure.

2.3 Final Exam

At the end of the course, students have been evaluated with a final exam. The final exam included questions about subtyping and CK machines[1]. The goal is not to test students on the language transformations per se, but rather on their understanding of subtyping and the CK machine. We therefore tested whether students would be able to use their understanding in practice. Our questions tested students on whether,

[1]The final exam also contained questions about other topics of the course. For example, the final exam of the second iteration contained questions about garbage collection. However, here we focus only on the parts of the exam that concern language transformations.

if presented with a language with unusual operators, they would be able to add subtyping to it, and derive its CK machine.

We have delivered two iterations of the course. The final exam took place online on both iterations due to the COVID-19 pandemic. In the first iteration of the course, we have shared a link to a text file that contained the content of the exam, and students submitted an updated text file via email. In the second iteration, the text of the exam was uploaded in the Blackboard system[2]. Students could insert their answers on the webpage as text, and submit them with the submit button.

The text of the final exam had two parts:

- The description of a toy language called **langFunny**.
- The questions that students were asked to answer, which referred to the language **langFunny**.

Below, we describe these two parts.

The Toy Language langFunny The text of the exam contained a description of **langFunny**. The text told the students that **langFunny** is a λ-calculus with pairs $\langle e_1, e_2\rangle$ and lists $[e_1, \ldots, e_n]$, equipped with two operators called **doublyApply** and **addToPairAsList**, which we describe below. The text of the exam did not repeat the typing rules and reduction rules of the λ-calculus with pairs and lists because we have seen them extensively in class, and because they did not play a role in the questions of the exam. On the contrary, the text of the exam provided the students with the formal semantics of **doublyApply** and **addToPairAsList**, which we will show shortly.

Below, we describe the operators **doublyApply** and **addToPairAsList**.

- **doublyApply**: The text of the final exam contained the following description of **doublyApply**. "**doublyApply** takes two functions f_1 and f_2 in input, and two arguments a_1 and a_2, and creates the pair $\langle f_2(f_1(a_1)), f_1(f_2(a_2))\rangle$. That is, the first component of the pair calls f_1 with a_1 and passes the result to f_2, and the second component calls f_2 with a_2 and passes the result to f_1."

  The text of the exam also provided the students with the following syntax, evaluation contexts, typing rule, and reduction rule for **doublyApply**.

$$\begin{array}{llll}
\text{Expression} & e & ::= & \ldots \mid (\textbf{doublyApply}\ e\ e\ e\ e) \\
\text{Evaluation Context} & E & ::= & \ldots \mid (\textbf{doublyApply}\ E\ e\ e\ e) \mid (\textbf{doublyApply}\ v\ E\ e\ e) \\
 & & & \mid (\textbf{doublyApply}\ v\ v\ E\ e) \mid (\textbf{doublyApply}\ v\ v\ v\ E)
\end{array}$$

$$\frac{\begin{array}{cc}\Gamma \vdash e_1 : T_1 \to T_2 & \Gamma \vdash e_2 : T_2 \to T_1 \\ \Gamma \vdash e_3 : T_1 & \Gamma \vdash e_4 : T_2\end{array}}{\Gamma \vdash (\textbf{doublyApply}\ e_1\ e_2\ e_3\ e_4) : T_2 \times T_1}$$

$$\textbf{doublyApply}\ v_1\ v_2\ v_3\ v_4 \longrightarrow \langle (v_2\ (v_1\ v_3)), (v_1\ (v_2\ v_4))\rangle$$

- **addToPairAsList**: The text of the exam contained the following description of **addToPairAsList**. "**addToPairAsList** takes an element a_1 and a pair p, and strives to add the element to the pair. As pairs contain only two elements, it creates a list with three elements: the element a_1, the first component of p, and the second component of p." To make a concrete example, we have **addToPairAsList** $a_1\ \langle a_2, a_3\rangle = [a_1, a_2, a_3]$.

[2]`https://www.blackboard.com/`

The text of the exam also provided the students with the following syntax, evaluation contexts, typing rule, and reduction rule for **addToPairAsList**.

$$\begin{array}{llll} \text{Expression} & e & ::= & \ldots \mid (\textbf{addToPairAsList}\; e\; e) \\ \text{Evaluation Context} & E & ::= & \ldots \mid (\textbf{addToPairAsList}\; E\; e) \mid (\textbf{addToPairAsList}\; v\; E) \end{array}$$

$$\frac{\Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : T \times T}{\Gamma \vdash (\textbf{addToPairAsList}\; e_1\; e_2) : \mathit{List}\; T}$$

$$\textbf{addToPairAsList}\; v_1\; \langle v_2, v_3 \rangle \longrightarrow [v_1, v_2, v_3]$$

Although these operators are not extremely bizarre, it is unusual to see them as primitive operations.

Questions and their Challenges After the description of the language **langFunny**, the text of the exam gave the students three questions that they were asked to answer. We dub these questions "Subtyping of **doublyApply**", "Subtyping of **addToPairAsList**", and "CK for **doublyApply**", respectively.

The question "Subtyping of **doublyApply**" asked the students to show the version of the typing rule of **doublyApply** with subtyping. This task is not trivial because the typing rule of **doublyApply** has three occurrences of T_1, and one of them is in contravariant position, which is the input of a function. Therefore, the other two occurrences of T_1 must be subtypes of that. The same scenario occurs for T_2.

The correct answer to this question is the following:
(The output type of this typing rule is more restrictive than necessary. The output type could be adjusted by applying another procedure, but we have omitted this part).

$$\frac{\begin{array}{cc} \Gamma \vdash e_1 : T_1 \to T_2' & \Gamma \vdash e_2 : T_2 \to T_1' \\ \Gamma \vdash e_3 : T_1'' & \Gamma \vdash e_4 : T_2'' \\ T_1' <: T_1 \quad T_1'' <: T_1 & T_2' <: T_2 \quad T_2'' <: T_2 \end{array}}{\Gamma \vdash (\textbf{doublyApply}\; e_1\; e_2\; e_3\; e_4) : T_2 \times T_1}$$

The question "Subtyping of **addToPairAsList**" asked the students to show the typing rule of the operator **addToPairAsList** with subtyping added. This task is also non-trivial because there are three occurrences of T that are peers. Therefore, the correct solution is to compute a join type.

The correct answer to this question is the following:

$$\frac{\begin{array}{c} \Gamma \vdash e_1 : T' \qquad \Gamma \vdash e_2 : T'' \times T''' \\ T = T' \vee T'' \vee T''' \end{array}}{\Gamma \vdash (\textbf{addToPairAsList}\; e_1\; e_2) : \mathit{List}\; T}$$

The question "CK for **doublyApply**" asked the students to derive the CK machine for **langFunny** insofar as the reduction rules for **doublyApply** are concerned. This operator is challenging because it has a high number of arguments (four). To complete the task, students must understand well the relationship between continuations and the evaluation order of arguments.

The correct answer to this question is the following:

$$\begin{array}{rll}
(\textbf{doublyApply}\ e_1\ e_2\ e_3\ e_4), k \longrightarrow e_1, (\textbf{doublyApply}_1\ e_2\ e_3\ e_4\ k) & \texttt{Start} \\
v_1, (\textbf{doublyApply}_1\ e_2\ e_3\ e_4\ k) \longrightarrow e_2, (\textbf{doublyApply}_2\ v_1\ e_3\ e_4\ k) & \texttt{Order} \\
v_2, (\textbf{doublyApply}_2\ v_1\ e_3\ e_4\ k) \longrightarrow e_3, (\textbf{doublyApply}_3\ v_1\ v_2\ e_4\ k) & \texttt{Order} \\
v_3, (\textbf{doublyApply}_3\ v_1\ v_2\ e_4\ k) \longrightarrow e_4, (\textbf{doublyApply}_4\ v_1\ v_2\ v_3\ k) & \texttt{Order} \\
v_4, (\textbf{doublyApply}_4\ v_1\ v_2\ v_3\ k) \longrightarrow \langle (v_2\ (v_1\ v_3)), (v_1\ (v_2\ v_4)) \rangle, k & \texttt{Computation}
\end{array}$$

The exam could also ask for the CK reduction rules of **addToPairAsList**. However, this task is slightly simpler than **doublyApply**, and we therefore were not interested in requesting those rules.

3 Evaluation

As we have previously said, we have run two iterations of the undergraduate PL course that we have described. To evaluate the merits of our thesis, we have collected information about students' success with the final exam, and more specifically, with their success in answering the questions "Subtyping of **doublyApply**", "Subtyping of **addToPairAsList**", and "CK for **doublyApply**".

For each question, we have evaluated the answer of each student as "Correct", "Partially Correct", "Partially Incorrect", and "Incorrect/Missing". Students' answers were classified as "Correct" only if they matched the solution given in the previous section. Answers were classified as "Incorrect/Missing" if they were missing, or they were completely incorrect. What constitutes a completely incorrect, a partially incorrect, and a partially correct answer is subjective by nature, therefore we have to draw a line in the sand, somehow subjectively. Our rationale is the following. A "Partially Correct" answer does not match the solution but shows that the student was on the way towards a correct solution. A "Partially Incorrect" answer contains some elements that demonstrates that the student is applying some correct reasoning principles. A completely incorrect answer ("Incorrect/Missing") provides no indication that the student is applying correct reasoning principles.

In total, we have conducted the study on 55 students. The rating of students' answers is shown in the following table.

	Correct	Partially Correct	Partially Incorrect	Incorrect/Missing
Subtyping of **doublyApply**	15	11	15	14
Subtyping of **addToPairAsList**	22	10	11	12
CK for **doublyApply**	18	13	10	14

The question "Subtyping of **doublyApply**" seems to be the most difficult among the three, as shown by the lowest number of completely correct answers. Subtyping of **doublyApply** is indeed a rather complicated task, as it involves contravariance. Furthermore, many variables are around, and a good number of them need to be subtype of a same variable. It is not surprising that 29 out of 55 did not provide a good answer (and were "Partially Incorrect" or "Incorrect/Missing"). On the contrary, it is rather encouraging to see that 26 out of 55 students could provide a good answer ("Correct" or "Partially Correct").

The question “Subtyping of **addToPairAsList**” seems to be the easiest among the three, as the highest number of students could provide a completely correct answer. Students could detect more easily that types are treated as peers in the typing rule of **addToPairAsList**, and perhaps this signals that this case is simpler to grasp than the contravariant case of **doublyApply**.

We are surprised by the results of the question “CK for **doublyApply**”, as such a machine seems to be rather involved. Regardless, a good number of students (18) could provide a completely correct answer, and a high number of them could give a good answer (“Correct” or “Partially Correct”). This is indicative that students could grasp the mechanics of the evaluation order, and translate it well as a CK machine.

It is safe to imagine that most students have not been exposed to formal semantics until this very course, and these questions are generally hard for them. It is encouraging to see that a good number of students could provide good answers. It may be an indication that, by and large, students could gain an understanding of subtyping and CK machines. However, we would like to explicitly say that we do not draw any general conclusion from this data.

A Note on Correctness As we have previously said, we do not claim any theoretical results of correctness of the algorithms that we have taught in class. However, we have implemented them as tools ([17]) that take a language specification in input written as a textual representation of operational semantics (with syntax similar to that of Ott [20]), and output the modified language specification (in the same textual format). We have applied these tools to several functional languages in order to add subtyping to them and derive their CK machines, and we have confirmed by inspecting the output languages that we have obtained the correct formulations.

3.1 Threats to Validity

The following observations keep this paper from drawing general conclusions about the thesis that language transformations are beneficial in class.

Further Studies While 55 students is a decent number, we would like to conduct more iterations of the same course, and have a larger pool of participants. When more data will be gathered, we plan to report on such data in a journal version of this paper.

Negative Experiments? It would be interesting to run instances of the course with language transformations, and also run instances *without* language transformations, while keeping the same syllabus and the final exam. The goal is to see whether there is a significant improvement in the success rate of exams in those courses that have used language transformations. However, we find pedagogical issues in implementing this plan. We think that adopting the final exam of Section 2.3 without having taught language transformations may not be a sensible choice. For example, simply covering subtyping with TAPL may not provide students with sufficient knowledge to complete the exam, and we may put unrealistic expectations on students’ ability to generalize and extrapolate general programming languages principles at the undergraduate level.

Perceived Effectiveness? We have made an attempt to evaluate whether students perceived that using language transformations was helpful for their learning. At the end of the course, we have given a survey for them to fill in. The survey contained six statements which, as typical in surveys, required a rating.

For example, to evaluate the task for “Subtyping of **doublyApply**”, the survey had the statements: “The language transformation algorithm for adding subtyping to languages helped me understand subtyping better”, and “The language transformation algorithm for adding subtyping to languages helped me add subtyping to the language at hand during the exam”. Students could assign a grade among “Strongly Agree”, “Somewhat Agree”, “Neither Agree nor Disagree”, “Somewhat Disagree” or “Strongly Disagree” to the statement. The survey requested students to rate the equivalent statements for “Subtyping of **addToPairAsList**” and “CK for **doublyApply**”.

Unfortunately, the survey did not receive participation. Our courses have taken place virtually during the COVID-19 pandemic, which may have been the cause of the experienced lack in participation.

4 Future Work

In this section, we discuss our plans. Our first goal is to evaluate the perceived effectiveness of language transformations with the survey that we have just described. Hopefully, participation to the survey will improve in the future. Other venues for future work are the following.

Improving our Current Language Transformations The procedures of Section 2.2 produce CK machines without environments, which is not typical. Our next step is to extend our procedures to capture the full Felleisen and Friedman’s CEK machine. Similarly, we plan on developing language transformations to automatically derive other popular abstract machines such as Landin’s SECD [16], and Krivine’s KAM [15] machines.

The language transformation that we have used for subtyping works only on simple types (sums, products, options, etcetera). We would like to extend the algorithm to capture also constructors that carry maps, such as records and objects. Maps can associate field names to values, and method names to functions. Maps come with their own subtyping properties such as width-subtyping, and permutation [19], and we plan on extending our algorithm to cover them. Similarly, we would like to develop language transformations for automatically adding bounded polymorphism [6, 1], recursive subtyping [2], multiple inheritance and mixins [4, 5] to languages, to make a few examples.

Language Transformations for Other Features Subtyping and abstract machines are not the only features that can be taught in a course in the principles of programming languages. We plan on addressing other features with language transformations, and using them in class.

In teaching the formalism of operational semantics, instructors may begin with a small-step or with a big-step semantics style. Whichever style has been chosen, it could be beneficial to explain the other style with language transformations (in addition to the planned material) that turn small-step into big-step, or big-step into small-step, respectively. Much work has been done to translate one style into the other [9, 11, 12, 13][3], and we plan on building upon this work. It is worth noting that converting from one style to the other may come with limitations. For example, it may not be possible to derive an equivalent big-step semantics from a small-step semantics formulation when parallelism is involved. The mentioned translation methods are subject to these limitations, and so will our corresponding language transformations.

We would like to develop a language transformation for automatically generating Milner-style type inference procedures. Also, we would like to devise a language transformation for adding generic types

[3]Among these works, [9] seems to be the most suitable for language transformations.

to languages. Some courses teach dynamic typing and run-time checking in some detail. We would like to explore the idea of automatically generating the dynamic semantics of dynamically typed languages based on a given type system. That is, a language transformation which relies on the type system to inform how the dynamic semantics should be modified in order to perform run-time type checking.

We are not aware of any work that automates the adding of the latter three examples. Developing such language transformations may be challenging research questions on their own.

Advanced Tasks Language transformations may be integrated in graduate level courses, as well. Some of these courses have a research-oriented flavour. In such courses, instructors may assign advanced tasks with language transformations. For example, instructors may ask students to study the work of Danvy et al. [11, 12, 13] to derive reduction semantics. The approach is rather elaborate, and involves techniques such as refocusing and transition compression. Instructors may ask students to develop a series of language transformations that capture this method. Similarly, instructors may ask students to model the language transformation for generating the pretty-big-step semantics from a small-step semantics [3]. Another idea is to target the Gradualizer papers [7, 8], for automatically adding gradual typing to languages.

5 Conclusion

Instructors can integrate language transformations into their undergraduate PL courses. We do not advocate replacing material, but to use language transformations in addition to the planned material. Our thesis is that language transformations are beneficial for students to help them deepen their understanding of the PL features being taught. In this paper, we have presented the study that we have conducted, and the results from this study. Although we refrain from declaring language transformations unequivocally beneficial, our numbers are encouraging, and we also offer this paper to open a conversation on the topic, and to inspire similar studies towards gathering evidence for, or against, our thesis.

Acknowledgements We would like to thank our EXPRESS/SOS 2021 reviewers for their feedback, which helped improve this paper.

References

[1] Martín Abadi & Luca Cardelli (1996): *A Theory of Objects*, 2nd edition. Monographs in Computer Science, Springer-Verlag, doi:10.1007/978-1-4419-8598-9.

[2] Roberto M. Amadio & Luca Cardelli (1993): *Subtyping Recursive Types*. *ACM Trans. Program. Lang. Syst.* 15(4), pp. 575–631, doi:10.1145/155183.155231.

[3] Casper Bach Poulsen & Peter D. Mosses (2014): *Deriving Pretty-Big-Step Semantics from Small-Step Semantics*. In: *Proceedings of the 23rd European Symposium on Programming Languages and Systems*, 8410, Springer-Verlag, Berlin, Heidelberg, pp. 270–289, doi:10.1007/978-3-642-54833-8_15.

[4] Gilad Bracha & William Cook (1990): *Mixin-Based Inheritance*. In: *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA/ECOOP '90, Association for Computing Machinery, New York, NY, USA, pp. 303–311, doi:10.1145/97945.97982.

[5] Luca Cardelli (1988): *A Semantics of Multiple Inheritance*. *Information and Computation* 76(2/3), pp. 138–164, doi:10.1016/0890-5401(88)90007-7.

[6] Luca Cardelli, John C. Mitchell, Simone Martini & Andre Scedrov (1994): *An Extension of System F with Subtyping*. *Information and Computation* 109(1/2), pp. 4–56, doi:10.1006/inco.1994.1013.

[7] Matteo Cimini & Jeremy G. Siek (2016): *The Gradualizer: A Methodology and Algorithm for Generating Gradual Type Systems*. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, Association for Computing Machinery, New York, NY, USA, pp. 443–455, doi:10.1145/2837614.2837632.

[8] Matteo Cimini & Jeremy G. Siek (2017): *Automatically Generating the Dynamic Semantics of Gradually Typed Languages*. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, ACM, New York, NY, USA, pp. 789–803, doi:10.1145/3093333.3009863.

[9] Ştefan Ciobâcă (2013): *From Small-Step Semantics to Big-Step Semantics, Automatically*. In: *Integrated Formal Methods, 10th International Conference, IFM 2013, Turku, Finland, June 10-14, 2013. Proceedings*, pp. 347–361, doi:10.1007/978-3-642-38613-8_24.

[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein (2009): *Introduction to Algorithms*, 3rd edition. The MIT Press.

[11] Olivier Danvy (2005): *From Reduction-based to Reduction-free Normalization*. *Electronic Notes in Theoretical Computer Science* 124(2), pp. 79–100, doi:10.1016/j.entcs.2005.01.007.

[12] Olivier Danvy (2008): *Defunctionalized Interpreters for Programming Languages*. In: *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, ACM, New York, NY, USA, pp. 131–142, doi:10.1145/1411204.1411206.

[13] Olivier Danvy & Lasse R. Nielsen (2004): *Refocusing in Reduction Semantics*. *BRICS Report Series* 11(26), doi:10.7146/brics.v11i26.21851.

[14] Matthias Felleisen & Matthew Flatt (2006): *Programming Languages and Lambda Calculi*. Notes available at https://www.cs.utah.edu/~mflatt/past-courses/cs7520/public_html/s06/notes.pdf and last accessed in August 2021.

[15] Jean-Louis Krivine (2007): *A Call-by-Name Lambda-Calculus Machine*. *Higher-Order and Symbolic Computation* 20(3), pp. 199–207, doi:10.1007/s10990-007-9018-9.

[16] Peter J. Landin (1965): *Correspondence Between ALGOL 60 and Church's Lambda-Notation: Part I*. *Communications of the ACM* 8, pp. 89–101, doi:10.1145/363744.363749.

[17] Benjamin Mourad (2019): *Lang-n-Change Tool*. https://github.com/bmourad01/lang-n-change.

[18] Benjamin Mourad & Matteo Cimini (2020): *A Calculus for Language Transformations*. In: *46th International Conference on Current Trends in Theory and Practice of Informatics (SOFSEM 2020)*, Springer, pp. 547–555, doi:10.1007/978-3-030-38919-2_44.

[19] Benjamin C. Pierce (2002): *Types and Programming Languages*, 1st edition. The MIT Press.

[20] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar & Rok Strniša (2007): *Ott: Effective Tool Support for the Working Semanticist*. In: *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, ICFP '07, ACM, New York, NY, USA, pp. 1–12, doi:10.1145/1291151.1291155.