The Gradualizer: A Methodology and Algorithm for Generating Gradual Type Systems

Matteo Cimini   Jeremy G. Siek
Indiana University, USA
mcimini@indiana.edu   jsiek@indiana.edu

Abstract

Many languages are beginning to integrate dynamic and static typing. Siek and Taha offered gradual typing as an approach to this integration that provides a coherent and full-span migration between the two disciplines. However, the literature lacks a general methodology for designing gradually typed languages. Our first contribution is to provide a methodology for deriving the gradual type system and the compilation to the cast calculus.

Based on this methodology, we present *the Gradualizer*, an algorithm that generates a gradual type system from a well-formed type system and also generates a compiler to the cast calculus. Our algorithm handles a large class of type systems and generates systems that are correct with respect to the formal criteria of gradual typing. We also report on an implementation of the Gradualizer that takes a type system expressed in lambda-prolog and outputs its gradually typed version and a compiler to the cast calculus in lambda-prolog.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Procedures, functions, and subroutines

General Terms Languages, Theory

Keywords gradual typing, type systems, semantics, methodology

1. Introduction

Many languages such as C# [5], Dart [32], Pyret, Racket [30, 31, 34, 35], TypeScript [6, 12] and VB [17] are beginning to integrate static and dynamic typing. Siek and Taha [22] created an approach, called *gradual typing*, that puts the programmer in control of which typing discipline is used for each region of code, provides seamless interoperability, and enables the convenient evolution of code from dynamic to static.

However, designing gradually typed languages has proven to be difficult [28]. Once we have a static type system in hand, there are two main questions: *how do I convert my statically typed language to gradual typing?* and *how do I know that I have designed a good gradually typed language?* To help language designers, our ongoing research program aims to provide formal correctness criteria, methodologies, and tools for supporting the shift to gradual typing.

POPL'16, January 20–22, 2016, St. Petersburg, FL, USA
© 2016 ACM. 978-1-4503-3549-2/16/01...$15.00
http://dx.doi.org/10.1145/2837614.2837632

This paper provides a general methodology and algorithm for deriving gradual type systems and deriving a compiler to the cast calculus. These together form the static aspects of a gradually typed language. We will report on a methodology for designing the runtime aspects of gradual typing in a future paper.

The literature lacks a general methodology for deriving a gradually typed calculus from a static one; the main resources available are the examples of typed calculi that are gradualized by experts [2, 4, 8, 14, 15, 22, 23, 27, 33, 34, 36, 38]. These papers serve as a reference but language designers are ultimately left without a disciplined approach on how to gradualize a new typed calculus.

Let us consider the simply typed λ-calculus (STLC) and its gradually typed version (GTLC). The GTLC adapts the STLC typing rule for function application in the following way.

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : T_{11} \to T_{12} \\ \Gamma \vdash e_2 : T_{11}\end{array}}{\Gamma \vdash e_1\ e_2 : T_{12}} \quad \Longrightarrow \quad \begin{array}{c} \dfrac{\begin{array}{c}\Gamma \vdash_G e_1 : \star \\ \Gamma \vdash_G e_2 : T_{11}\end{array}}{\Gamma \vdash_G e_1\ e_2 : \star} \\[2em] \dfrac{\begin{array}{c}\Gamma \vdash_G e_1 : T_{11} \to T_{12} \\ \Gamma \vdash_G e_2 : T_2 \quad T_{11} \sim T_2\end{array}}{\Gamma \vdash_G e_1\ e_2 : T_{12}} \end{array}$$

This example reveals some useful patterns. For instance, the first application rule of the GTLC replaces the function type with the unknown type $\star$ (aka. the dynamic type). In general, gradual type systems should allow any subexpression to have type $\star$, but $\star$ would fail a syntactic pattern-match with the form $T_{11} \to T_{12}$ of the original rule. So we see that occurrences of *constructed types*, that is, types that are applications of some type constructor, may need such special treatment. However, not every occurrence of a constructed type requires this treatment. Consider the STLC typing rule for abstraction and the corresponding rule in the GTLC.

$$\frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash (\lambda x{:}T_1.\, e) : T_1 \to T_2} \quad \Longrightarrow \quad \frac{\Gamma, x : T_1 \vdash_G e : T_2}{\Gamma \vdash_G (\lambda x{:}T_1.\, e) : T_1 \to T_2}$$

The rule remains unchanged in its gradual counterpart, so we are left wondering why the constructed type $T_1 \to T_2$ in the conclusion of the rule does not need special treatment.

Returning to the second application rule of GTLC, we see the tagline of gradual typing: *replace type equality with consistency* ($\sim$) [23]. However, consider the typing rule for sum types and a hypothetical rule for sums in a gradual type system.

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : T_{11} + T_{12} \\ \Gamma, x : T_{11} \vdash e_2 : T \qquad \Gamma, y : T_{12} \vdash e_3 : T\end{array}}{\Gamma \vdash (\texttt{case}\, e_1\, \texttt{of}\, \texttt{inl}\, x \Rightarrow e_2 \mid \texttt{inr}\, y \Rightarrow e_3) : T}$$

$$\Downarrow$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : T_{11} + T_{12} \\ \Gamma, x : T_{11} \vdash e_2 : T' \quad \Gamma, y : T_{12} \vdash e_3 : T'' \quad T = T' \sqcup T''\end{array}}{\Gamma \vdash (\texttt{case}\, e_1\, \texttt{of}\, \texttt{inl}\, x \Rightarrow e_2 \mid \texttt{inr}\, y \Rightarrow e_3) : T}$$

Type equality is used multiple times in the original rule: we have two occurrences of T_{11}, two of T_{12}, and three of T. However, only the two highlighted occurrences of T need special treatment in the gradually typed version. Further, instead of using consistency, the gradually typed version takes the *join* of the types of e_2 and e_3 to be the type of the entire `case` expression.

The designer of a gradual type system implicitly uses knowledge about the operational semantics of the language regarding the *flow* of values to make choices such as how to treat the variable T above. For the `case` expression, the value of e_2 or of e_3 may become the result of the whole expression, so the type of the whole expression needs to be consistent with both branches. Hence the use of the join operation.

The designer of a gradual type system also uses knowledge about the input/output modes (in the sense of logic programming) of the typing judgment to make decisions. For a simple typing judgment of the form $\Gamma \vdash e : T$, the environment Γ and expression e are typically inputs and the type T is an output. For the `case` expression, we do not need to relate the two occurrences of variable T_1 with consistency (or join) because one occurrence is an output and the other is an input. It is only when two or more occurrences of a variable are outputs that the gradual version of the typing rule needs to use consistency or join.

The dynamic semantics of a gradually typed language is given by a type-directed translation to a cast calculus. This translation should be type preserving, so the designer must keep in mind the type system of the cast calculus. Here we make the (typical) assumption that the cast calculus has the same fully static type system as the input language except that it adds an expression for explicit casts. The translation to the cast calculus is type directed and therefore tightly connected to the gradual type system, so this all has bearing on the design of the gradual type system. Consider the typing rule for `cons` from *Types and Programming Languages* [19]:

$$\frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : \texttt{list}\, T}{\Gamma \vdash \texttt{cons}[T]\, e_1\, e_2 : \texttt{list}\, T}$$

In the gradually typed version, we would have e_1 at type T' and e_2 at type $\texttt{list}\, T''$, but how should T' and T'' be related and what should the type of the whole `cons` expression be? The cast calculus uses the above rule for `cons`, and the type T appears as an annotation on `cons`. Type annotations should not be changed by the translation, as that could distort the programmer's intent, so the type of the whole expression must be $\texttt{list}\, T$ and the types of the two arguments must be T and $\texttt{list}\, T$. So we need to cast T' to T and $\texttt{list}\, T''$ to $\texttt{list}\, T$. Thus, one of the gradually typed versions of this rule must be as follows. (The other rule has e_2 at type $\star$.)

$$\frac{\Gamma \vdash e_1 : T' \quad T' \sim T \quad \Gamma \vdash e_2 : \texttt{list}\, T'' \quad T'' \sim T}{\Gamma \vdash \texttt{cons}[T]\, e_1\, e_2 : \texttt{list}\, T}$$

In this paper, we develop a unified methodology for generating gradual type systems based on the above observations. We walk the reader through the steps of turning a static type system into a gradual type system and also of deriving the compilation to the cast calculus. This part of the paper will be tutorial in style, as we hope it will serve as a useful reference for language designers.

The Gradualizer In the latter part of the paper we make our methodology precise and automatic in the form of the *Gradualizer*, a procedure that takes a type system as input, represented as a logic program in λ-prolog, and produces the gradually typed version of it and the compilation procedure to the cast calculus. The Gradualizer automatically applies our methodology by transforming logic programs. We have implemented the Gradualizer in Haskell. Figure 1 shows an example input and the output of the Gradualizer. The `typeof` predicate expresses the type system of the STLC and is an example input. The predicates `typeofG` and `typeofCC` are generated by the Gradualizer. The former is the type system for the GTLC and the latter is the type system for the cast calculus. The procedure also produces the compiler to the cast calculus `compToCC`.

The Gradualizer procedure that we define is formal enough to be the subject of proofs. Indeed, our methodology and the systems generated by the Gradualizer are useful only so long as they are *correct*. Siek et al. [21] recently expanded and refined the correctness criteria for a gradual type system, so we have precise standards to meet. We prove that the Gradualizer always generates typed calculi that satisfy the criteria that apply to the static aspects of gradual typing. This validates our methodology and provides high confidence in using the Gradualizer and its implementation.

In summary, this paper makes the following contributions.

1. A methodology for generating the static semantics of gradually-typed languages. For the first time, we give an explicit walk-through of the process with a degree of generality that includes a large class of type systems (Section 3).
2. We show the applicability of our methodology by gradualizing a number of type systems, mostly from Pierce [19]: STLC, unit type, pairs, tuples, let binding, let rec binding, general recursion (fix), sum types, exceptions, references, lists, if-then-else, and STLC with integers and addition (Section 4).
3. The Gradualizer: an algorithm for generating a gradual type system and a compiler from a static type system (Section 6). We provide an implementation of the Gradualizer in Haskell, which produces type checkers and compilers in λ-prolog.
4. We validate our methodology by proving that the systems generated by the Gradualizer always satisfy the formal correctness criteria for gradual typing (Section 7).

The implementation of the Gradualizer can be found at the following github repository:
`https://github.com/mcimini/Gradualizer`.

2. Overview of Gradual Typing

A gradually typed language involves three languages: the gradually typed language itself, a statically typed language that it is gradual with respect to, and a cast calculus that specifies the runtime semantics of casts. In this section we review the prototypical example: the Gradually Typed Lambda Calculus (GTLC), the Simply Typed Lambda Calculus (STLC), and the Cast Calculus (CC)[1].

Simply Typed Lambda Calculus (STLC) For ease of reference, we reproduce the standard syntax and type system for the STLC in Figure 2. (The type `Bool` here simply serves as a base case.)

Syntax

$$\begin{array}{lll} \text{Types } T & ::= & T \rightarrow T \mid \texttt{Bool} \\ \text{Terms } e & ::= & x \mid \lambda x{:}T.\, e \mid e\ e \end{array}$$

$\boxed{\Gamma \vdash e : T}$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{\Gamma, x : T_1 \vdash e : T_2}{\Gamma \vdash (\lambda x{:}T_1.\, e) : T_1 \rightarrow T_2}$$

$$\frac{\Gamma \vdash e_1 : T_{11} \rightarrow T_{12} \quad \Gamma \vdash e_2 : T_{11}}{\Gamma \vdash e_1\ e_2 : T_{12}}$$

Figure 2. The Simply Typed Lambda Calculus (STLC).

[1] This cast calculus is closely related to the Blame Calculus [37].

```
typeof (abs T1 R) (arrow T1 T2) :- (pi x\ (typeof x T1 => typeof (R x) T2)).
typeof (app E1 E2) T2 :- typeof E1 (arrow T1 T2), typeof E2 T1.

typeofG (abs T1 E) (arrow T1 T2) :- (pi x\ (typeofG x T1 => typeofG (E x) T2)).
typeofG (app E1 E2) T2 :- typeofG E1 PM1, matchArrow PM1 T1 T2,
                          typeofG E2 New1, consistency New1 T1.

typeofCC (abs T1 E) (arrow T1 T2) :- (pi x\ (typeofCC x T1 => typeofCC (E x) T2)).
typeofCC (app E1 E2) T2 :- typeofCC E1 (arrow T1 T2), typeofCC E2 T1.
typeofCC (cast E T1 L T2) T2 :- typeofCC E T1.

compToCC (abs T1 E) (abs T1 (x\ (E' x))) (arrow T1 T2) :-
                          (pi x\ (compToCC x x T1 => compToCC (E x) (E' x) T2)).
compToCC (app E1 E2) (app (cast E1' PM1 L (arrow T1 T2)) (cast E2' New1 L T1)) T2 :-
                          compToCC E1 E1' PM1, matchArrow PM1 T1 T2,
                          compToCC E2 E2' New1, consistency New1 T1.
```

Figure 1. Example input (`typeof`) and outputs (`typeofG`, `typeofCC`, `compToCC`) of the Gradualizer.

Syntax

$$\begin{array}{lll} \text{Types } T & ::= & T \to T \mid \texttt{Bool} \mid \star \\ \text{Terms } e & ::= & x \mid \lambda x{:}T.\, e \mid e\, e \end{array}$$

$\boxed{\Gamma \vdash_G e : T}$

$$\frac{x : T \in \Gamma}{\Gamma \vdash_G x : T} \qquad \frac{\Gamma, x : T_1 \vdash_G e : T_2}{\Gamma \vdash_G (\lambda x{:}T_1.\, e) : T_1 {\to} T_2}$$

$$\frac{\begin{array}{cc} \Gamma \vdash_G e_1 : T_1 & T_1 \rhd (T_{11} \to T_{12}) \\ \Gamma \vdash_G e_2 : T_2 & T_2 \sim T_{11} \end{array}}{\Gamma \vdash_G e_1\, e_2 : T_{12}}$$

$\boxed{T \sim T}$ Consistency

$$T \sim \star \qquad \star \sim T \qquad \frac{T_1 \sim T_3 \qquad T_2 \sim T_4}{T_1 \to T_2 \sim T_3 \to T_4}$$

$\boxed{T \rhd T}$ Matching

$$(T_1 \to T_2) \rhd T_1 \to T_2 \qquad \star \rhd \star \to \star$$

Figure 3. The Gradually Typed Lambda Calculus (GTLC).

Gradually Typed Lambda Calculus (GTLC) The GTLC has the same terms as the STLC but extends the types with the unknown type $\star$ and adjusts the typing rules with a proper treatment of $\star$. For example, a function that accepts an argument of type $\star$ should accepts integer arguments, as in $(\lambda x{:}\star.\, x)\ 4$. On the other hand, a gradual type system should still reject programs with obvious static type errors, such as $(\lambda x : \texttt{Bool}.\, x)\ 4$. Gradual typing achieves both of these aims with the help of the consistency relation on types, written $T_1 \sim T_2$. Figure 3 shows the syntax and typing rules for the GTLC and defines the consistency relation.

The formulation of Figure 3 differs from the original presentation of the GTLC [22]. Indeed, it reflects the style adopted in recent work in gradual typing [10, 21, 27]. This formulation avoids the duplication of the typing rule for function application through the use of a pattern-matching relation on types, also defined in Figure 3. In particular, we pattern match the type T_1 with a function type $T_{11}{\to}T_{12}$. Referring to the definition of matching ($\rhd$), we see that if T_1 is a function type, then T_{11} and T_{12} are its domain and codomain. On the other hand, if T_1 is $\star$, then the second clause of the definition of $\rhd$ realizes the well-known behavior for gradual typing: *the occurrence of $\star$ in lieu of a function type must be treated as $\star \to \star$*. If T_1 is any other type, then $T_1 \rhd (T_{11}{\to}T_{12})$ is false and the typing rule for application is not applicable.

Syntax

$$\begin{array}{lll} \text{Types } T & ::= & \ldots \mid \star \\ \text{Terms } e & ::= & \ldots \mid e : T \Rightarrow^{\ell} T \end{array}$$

$\boxed{\Gamma \vdash_{CC} e : T}$

$$\ldots \qquad \frac{\Gamma \vdash_{CC} e : T_1}{\Gamma \vdash_{CC} (e : T_1 \Rightarrow^{\ell} T_2) : T_2}$$

Figure 4. The Cast Calculus (CC) extends the STLC.

$\boxed{\Gamma \vdash_{CC} e \rightsquigarrow e' : T}$

$$\frac{x : T \in \Gamma}{\Gamma \vdash_{CC} x \rightsquigarrow x : T}$$

$$\frac{\Gamma, x : T_1 \vdash_{CC} e \rightsquigarrow e' : T_2}{\Gamma \vdash_{CC} (\lambda x{:}T_1.\, e) \rightsquigarrow (\lambda x{:}T_1.\, e') : T_1 {\to} T_2}$$

$$\frac{\begin{array}{cc} \Gamma \vdash_{CC} e_1 \rightsquigarrow e_1' : T_1 & T_1 \rhd T_{11}{\to}T_{12} \\ \Gamma \vdash_{CC} e_2 \rightsquigarrow e_2' : T_2 & T_2 \sim T_{11} \end{array}}{\Gamma \vdash_{CC} e_1\, e_2 \rightsquigarrow (e_1' : T_1 \Rightarrow^{\ell_1} T_{11}{\to}T_{12})(e_2' : T_2 \Rightarrow^{\ell_2} T_{11}) : T_{12}}$$

Figure 5. Compilation of the GTLC to the CC.

The Cast Calculus (CC) The cast calculus serves to make explicit the implicit casts that are introduced by consistency and pattern matching in the GTLC. The operational semantics is then defined in terms of the explicit casts. Formally, the cast calculus, shown in Figure 4, extends the STLC with the unknown type $\star$ and with a cast expression of the form $e : T_1 \Rightarrow^{\ell} T_2$ [3], where T_1 is the static type of e, T_2 is the target type, and ℓ is a blame label.

Compilation of the GTLC to the CC The compilation to the Cast Calculus is written $\Gamma \vdash_{CC} e \rightsquigarrow e' : T$, meaning that e is compiled to e' and has type T in the type environment Γ.

This compilation, also known as *cast insertion*, inserts appropriate run-time casts at the points where the gradual type system uses consistency or pattern matching. Recall that consistency just says that the value *might* have the expected type, so a run-time check is necessary in the form of these casts. The compiler from the GTLC to the Cast Calculus is shown in Figure 5. The compilation judgment is exactly the same as the GTLC type system if one ignores the output expression.

$\boxed{T \sqsubseteq T}$

$$\frac{}{\star \sqsubseteq T} \qquad \frac{}{\texttt{Bool} \sqsubseteq \texttt{Bool}} \qquad \frac{T_1 \sqsubseteq T_3 \quad T_2 \sqsubseteq T_4}{T_1 \to T_2 \sqsubseteq T_3 \to T_4}$$

$\boxed{e \sqsubseteq e}$

$$\frac{}{x \sqsubseteq x} \qquad \frac{T_1 \sqsubseteq T_2 \quad e_1 \sqsubseteq e_2}{\lambda x{:}T_1.\, e_1 \sqsubseteq \lambda x{:}T_2.\, e_2} \qquad \frac{e_1 \sqsubseteq e_1' \quad e_2 \sqsubseteq e_2'}{(e_1\ e_2) \sqsubseteq (e_1'\ e_2')}$$

Figure 6. Type and Term Precision.

Correctness Criteria for Gradual Typing What are the properties that a gradually typed calculus must have? Recent work addresses this matter and puts forward a set of correctness criteria for gradual typing [21]. We review here the criteria that are relevant to this paper, those related to the static aspects of gradual typing.

Correctness Criteria [21]

Conservative extension:
for all static Γ, e and T, $\Gamma \vdash e : T$ iff $\Gamma \vdash_G e : T$.
Monotonicity w.r.t. precision:
for all Γ, e, e', T, if $\Gamma \vdash_G e : T$ and $e' \sqsubseteq e$,
then $\Gamma \vdash_G e' : T'$ and $T' \sqsubseteq T$ for some T'.
Type preservation of cast insertion:
for all Γ, e, T, if $\Gamma \vdash_G e : T$, then $\Gamma \vdash e \rightsquigarrow e' : T$
and $\Gamma \vdash_{CC} e' : T$ for some e'.
Monotonicity of cast insertion:
for all Γ, e_1, e_2, e_1', T, if $\Gamma \vdash_{CC} e_1 \rightsquigarrow e_1' : T$
and $\Gamma \vdash_{CC} e_2 \rightsquigarrow e_2' : T$ and $e_1 \sqsubseteq e_2$ then $e_1' \sqsubseteq e_2'$.

The first criterion imposes that typeability of $\vdash_G$ and $\vdash$ coincide over *static* programs, that is, programs that do not contain any $\star$. This criteria guarantees that well-typed programs of the original language remain well-typed in the gradually typed language. Furthermore, it ensures that ill-typed programs of the original language remain so in the gradually typed language.

The second criterion ensures that adding or removing appropriate type annotations does not cause a gradually typed program to become ill-typed. This criterion is expressed using the precision relation over types and programs, defined in Figure 6. In this paper we adopt the direction of the lattice in which $T_1 \sqsubseteq T_2$ denotes that T_1 is less precise than T_2, that is, the inverse of naive subtyping [37]. The precision relation over types lifts to programs, i.e. $e_1 \sqsubseteq e_2$ means that e_1 and e_2 are the same program except that e_1 makes use of less precise type annotations. The *join* operation, written $T_1 \sqcup T_2$, is the least upper bound with respect to the precision relation $\sqsubseteq$. The first two criteria are fundamental for gradual typing. They explain for instance why both the programs $(\lambda x : \texttt{Int})\ 4$ and $(\lambda x : \star.\, x)\ 4$ *must* be typeable in GTLC, as the former is typeable in STLC and the latter is a less-precise version of it.

The compilation to the cast calculus must also conform to some criteria. In particular, it must be total for typeable programs, it must be type preserving, and it must be monotonic over the precision relation $\sqsubseteq$. (For the definition of precision for the Cast Calculus, see Siek et al. [21].)

In Section 7 we prove that the Gradualizer always generates gradually typed calculi that satisfy all of these criteria.

3. A Methodology for Gradualizing Languages

The methodology for deriving the gradual type system consists of the following steps. We summarize the steps here and describe them in detail in the indicated subsections.

1. Classify input/output modes (Section 3.1).
2. Classify producer/consumer variables (Section 3.2).
3. Replace outputs that are constructed types with new variables and apply pattern matching (Section 3.3).
4. Replace output variables with fresh variables, mark each producer as flowing to consumers through their *final* type, and replace each input by its final type (Section 3.4).
5. Restrict lone input variables to range over static types only (Section 3.5).
6. Remove flows to join types and replace the remaining flow premises with consistency checks (Section 3.6).

Generating a compiler to the cast calculus is similar but requires an additional step between steps 5 and 6.

5.5. Generate casts as directed by the flow premises (Section 3.7).

3.1 Classify Input/Output Modes

Our methodology applies the notion of *input/output modes* to the typing relation and auxiliary relations such as subtyping. This notion stems from logic programming where a relation is attributed input and output parameters [7, 16, 20, 29]. In a correctly moded logic program, input arguments should be ground (or, instantiated) at the moment of a use of a predicate while outputs will become ground because of the use. So the outputs are a function (in the mathematical sense) of the inputs, i.e., they are determined by the inputs.

In a typical typing relation $\Gamma \vdash e : T$, the environment Γ and expression e are inputs and the type T is an output. As a convention, we color outputs in blue and inputs in red. Our methodology uses these modes to classify each occurrence of a variable in typing rules. Consider again the typing rule for case expressions.

$$\frac{\Gamma \vdash e_1 : T_{11} + T_{12} \qquad \Gamma, x : T_{11} \vdash e_2 : T \qquad \Gamma, y : T_{12} \vdash e_3 : T}{\Gamma \vdash (\texttt{case}\, e_1\ \texttt{of}\ \texttt{inl}\, x \Rightarrow e_2 \mid \texttt{inr}\, y \Rightarrow e_3) : T}$$

The occurrences of T_{11} and T_{12} in the premise $\Gamma \vdash e_1 : (T_{11} + T_{12})$ are classified as outputs, as are the occurrences of T in the premises for e_2 and e_3. On the other hand, the occurrences of T_{11} in the premise for e_2 and T_{12} in the premise for e_3 are classified as inputs. When classifying variables in the conclusion of the rule, the modes are flipped because the conclusion represents the input to the logic predicate (similar to the contravariance of a function). So the T in the conclusion is classified as an input.

3.2 Classify Producers and Consumers

As we discussed in Section 1, the operational semantics play a primary role in the design of the gradual typing system. For example, our treatment of type T in the above typing rule for `case` expressions depends on the operational semantics of `case`. Recall the following standard rules for reducing to either the left or right branch depending on tag on the sum.

$$(\texttt{case}\,(\texttt{inl}\, v)\ \texttt{of}\ \texttt{inl}\, x \Rightarrow e_2 \mid \texttt{inr}\, y \Rightarrow e_3) \longrightarrow [v/x]e_2$$
$$(\texttt{case}\,(\texttt{inr}\, v)\ \texttt{of}\ \texttt{inl}\, x \Rightarrow e_2 \mid \texttt{inr}\, y \Rightarrow e_3) \longrightarrow [v/y]e_3$$

These rules tell us that the value from e_2 or from e_3 can become the result for the `case` expression. Also, the rules tell us that the value v flows into e_2 or e_3 via substitution. Understanding the flow induced by the operational semantics is necessary for the correct design of a gradual type system. However, such flow information seems hard to detect in general and hard to recover automatically.

Fortunately, we have identified a simple heuristic that approximates this information. We classify each variable occurrence as being in either a *producer* or *consumer* position. This notion is similar but subtly different from that of the output/input modes we discussed above, and is based on the notion of positive and negative positions within a type [19].

The producer/consumer notion is similar to that of the input/output modes in that occurrences in output positions are classified as producers and occurrences in input positions are classified as consumers. In the below example we mark the variable occurrences with a 'p' for producer and 'c' for consumer.

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : T_{11}{}^{\mathsf{p}} + T_{12}{}^{\mathsf{p}} \\ \Gamma, x : T_{11}{}^{\mathsf{c}} \vdash e_2 : T^{\mathsf{p}} \qquad \Gamma, y : T_{12}{}^{\mathsf{c}} \vdash e_3 : T^{\mathsf{p}}\end{array}}{\Gamma \vdash (\texttt{case}\, e_1\, \texttt{of}\, \texttt{inl}\, x \Rightarrow e_2 \mid \texttt{inr}\, y \Rightarrow e_3) : T^{\mathsf{c}}}$$

The producer/consumer notion is different from that of input/output modes once we take into account the polarity of type constructors. For example, in a function type, the polarity flips in the domain type (it is contravariant). Consider the rule for function application. The type of e_1 is $T_{11}{\to}T_{12}$, which is in producer position, but the occurrence of T_{11} inside this type is a consumer because of the flip in polarity for the domain of a function.

$$\frac{\Gamma \vdash e_1 : T_{11}{}^{\mathsf{c}} \to T_{12}{}^{\mathsf{p}} \qquad \Gamma \vdash e_2 : T_{11}{}^{\mathsf{p}}}{\Gamma \vdash e_1\ e_2 : T_{12}{}^{\mathsf{c}}}$$

The distinction between producers and consumers determines flow as we shall see in Section 3.4. The general idea is that producers are connected to consumers with flow predicates. Thus, for function application, we have a flow from $T_{11}{}^{\mathsf{p}}$ to $T_{11}{}^{\mathsf{c}}$, which matches the the expected behavior of the operational semantics of application $(\lambda x.\, e)\ v \longrightarrow [v/x]e$, which entails that v flows into e.

Polarity also plays a role in the `case` expression above, but in a way that is hard to see. For sum types, there is no flip in polarity (they are covariant), so given that $T_{11} + T_{12}$ is in producer position, T_{11} and T_{12} are also in producer position.

3.3 Apply Pattern Matching to Constructed Outputs

We recall that constructed types are types that are not simply variables. When we use a typing judgment $\Gamma \vdash e : T$ as a premise in a typing rule, the type of e is determined and then assigned to the variable T. However, when a constructed type such as $T_{11}{\to}T_{12}$ is in the output position, then we have to apply pattern-matching to allow the subexpression to have type $\star$. In this step, we

> *Replace constructed outputs with fresh variables and pattern match these variables against the constructed outputs.*

As an example, we apply this step of the methodology to the typing rule for application of STLC. We start with the result of classifying the variables according to input/output and producer/consumer as discussed above.

$$\frac{\Gamma \vdash e_1 : T_{11}{}^{\mathsf{c}}{\to}T_{12}{}^{\mathsf{p}} \qquad \Gamma \vdash e_2 : T_{11}{}^{\mathsf{p}}}{\Gamma \vdash e_1\ e_2 : T_{12}}$$

The type $T_{11}{}^{\mathsf{c}}{\to}T_{12}{}^{\mathsf{p}}$ is an output that is a constructed type. Therefore, we replace the function type with a fresh variable T_1 (called a *pattern matching variable*) and pattern match it with the function type, producing the following rule.

$$\frac{\Gamma \vdash_G e_1 : T_1 \qquad T_1 \rhd T_{11}{}^{\mathsf{c}}{\to}T_{12}{}^{\mathsf{p}} \qquad \Gamma \vdash_G e_2 : T_{11}{}^{\mathsf{p}}}{\Gamma \vdash_G e_1\ e_2 : T_{12}{}^{\mathsf{c}}}$$

Thanks to the pattern-matching premise, the rule above handles the case where T_1 is instantiated with $\star$ (let us recall the definition clause $\star \rhd \star \to \star$). Were we not to handle this case, the program $\lambda x{:}\,\star\,.\,(x\ \ 42)$ would not be well-typed even though the more precise version $(\lambda x{:}\texttt{Int}{\to}\texttt{Int}.\, x\ \ 42)$ is well-typed. Therefore, monotonicity of typing w.r.t. the precision relation would fail to hold and the gradual type system would be incorrect.

We treat type constants such as `Int` and `Bool` as type constructors of arity 0. Consider the following example.

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \texttt{Int}^{\mathsf{p}} \\ \Gamma \vdash e_2 : \texttt{Int}^{\mathsf{p}}\end{array}}{\Gamma \vdash e_1 + e_2 : \texttt{Int}^{\mathsf{c}}} \Longrightarrow \frac{\begin{array}{cc}\Gamma \vdash e_1 : T_1 & T_1 \rhd \texttt{Int}^{\mathsf{p}} \\ \Gamma \vdash e_2 : T_2 & T_2 \rhd \texttt{Int}^{\mathsf{p}}\end{array}}{\Gamma \vdash e_1 + e_2 : \texttt{Int}^{\mathsf{c}}}$$

This typing rule is equivalent to the rule for $+$ in the literature [10]. The matching rules for `Int` are

$$\texttt{Int} \rhd \texttt{Int} \qquad \star \rhd \texttt{Int}$$

Types that are in the conclusion and in the type environment are sometimes constructed types. An example is in the object creation rule of the object calculus [1]. (In the rule, b is an expression, A and B are types, and E is the type environment.)

$$\frac{E, x_i{:}A^{\mathsf{c}} \vdash b_i : B_i{}^{\mathsf{p}} \qquad \forall i \in 1..n}{E \vdash [l_i{=}\varsigma(x_i{:}A^{\mathsf{p}})\, b_i{}^{\,i\in 1..n}] : A^{\mathsf{c}}} \quad \text{if } A \equiv [l_i{:}B_i{}^{\,i\in 1..n}]$$

Even though A is a constructed type (because $A \equiv [l_i{:}B_i{}^{\,i\in 1..n}]$), the red occurrences of A are inputs and therefore they do not require pattern matching. The typing rule for object creation remains unchanged in the gradual type system [23].

3.4 Flow and Final Type Discovery

As mentioned in Section 1, the design of a gradual type system must take into account that it guides the compilation to the Cast Calculus and that the compilation must be type preserving. Consider again the typing rule for `cons`.

$$\frac{\Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : \texttt{list}\, T}{\Gamma \vdash \texttt{cons}[T]\, e_1\, e_2 : \texttt{list}\, T}$$

The rule serves both as the input to the Gradualizer and also as the typing rule in the cast calculus. The following shows the result of applying the first three steps to this rule (classify input/output modes, classify producers and consumers, and pattern match constructed outputs). (These lists are immutable so the `list` type constructor is covariant.)

$$\frac{\Gamma \vdash e_1 : T^{\mathsf{p}} \qquad \Gamma \vdash e_2 : T_2 \qquad T_2 \rhd \texttt{list}\, T^{\mathsf{p}}}{\Gamma \vdash \texttt{cons}[T]\, e_1\, e_2 : \texttt{list}\, T^{\mathsf{c}}}$$

Now we are ready to describe the current step of the methodology. We start by replacing variables in output positions by distinct variables, which we refer to as different versions of the same type. We replace the three output occurrences of T by T, T', and T''.

$$\frac{\Gamma \vdash e_1 : T'^{\mathsf{p}} \qquad \Gamma \vdash e_2 : T_2 \qquad T_2 \rhd \texttt{list}\, T''^{\mathsf{p}}}{\Gamma \vdash \texttt{cons}[T]\, e_1\, e_2 : \texttt{list}\, T^{\mathsf{c}}}$$

Next we need to consider how the compilation to the cast calculus will preserve types. For the translation to be well typed, the first and second arguments to `cons` need to use the same type T, which also matches the type annotation T and the element type of the result `list` T. So we need to choose which version of the type should instantiate this type T. We refer to this type as the ***final type*** and use the following rules, in order, to choose it.

1. If the variable appears in an annotation, then it is the final type.
2. If the variable is an output consumer, then it is the final type.
3. Otherwise, the final type is the join of all the producers.

In the `cons` example, rule 1 applies, so we choose T (the occurrence in the annotation) as the final type for T and all its versions. We then connect the producers to the consumers by way of the final type. We do so using a flow relation written $T_1 \rightsquigarrow T_2$ and read T_1 *flows to* T_2. The meaning of a flow $T_1 \rightsquigarrow T_2$ is that T_1 is consistent with T_2. Moreover, the direction of the flow drives our methodology in inserting the correct casts in the cast insertion procedure (Section 3.7). The following are the steps for inserting flow information.

1. *Producers flow to their final type and final types flow to the consumers.*
2. *Variable occurrences in input positions are changed to their final type.*

The result for the `cons` example is shown below.

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : T' \quad \Gamma \vdash e_2 : T_2 \quad T_2 \rhd \mathtt{list}\, T'' \\ T' \leadsto T \qquad\qquad T'' \leadsto T\end{array}}{\Gamma \vdash \mathtt{cons}[T]\, e_1\, e_2 : \mathtt{list}\ T}$$

There are no consumers in the rule above and so there are no flows starting from T.

Consider the STLC application rule after we apply the treatment of this section.

$$\frac{\begin{array}{c}\Gamma \vdash_G e_1 : T_1 \qquad T_1 \rhd T_{11} \rightarrow T_{12} \\ \Gamma \vdash_G e_2 : T'_{11} \quad T'_{11} \leadsto T_{11}\end{array}}{\Gamma \vdash_G e_1\ e_2 : T_{12}}$$

The final type for the producer $T_{11}{}^{\mathrm{p}}$ (with new variable T'_{11}) is the consumer $T_{11}{}^{\mathrm{c}}$ (T_{11}), hence the flow premise $T'_{11} \leadsto T_{11}$. Because $T_{12}{}^{\mathrm{p}}$ is the only producer for T_{12}, its final type is the join of just T_{12}, that is itself. So the second step leaves the rule unchanged.

In the situation where a typing rule contains multiple output consumers for the same variable, there is no known way to create a gradually typed version of the rule that satisfies our criteria. Consider the following imaginary application operator `dapp` that takes two functions and one argument, then applies both functions to the argument and returns the pair of results.

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : T_1{}^{\mathrm{c}} \rightarrow T_2{}^{\mathrm{p}} \qquad \Gamma \vdash e_2 : T_1{}^{\mathrm{c}} \rightarrow T_3{}^{\mathrm{p}} \\ \Gamma \vdash e_3 : T_1{}^{\mathrm{p}}\end{array}}{\Gamma \vdash (\mathtt{dapp}\ e_1\ e_2\ e_3) : T_2{}^{\mathrm{c}} \times T_3{}^{\mathrm{c}}}$$

After we make the two output consumers $T_1{}^{\mathrm{c}}$ distinct, changing the second to T'_1, where should $T_1{}^{\mathrm{p}}$ flow to? We could have it flow into both. However, this solution causes a problem in cast insertion, as e_3 can have only one cast wrapped around it. Picking either one of them, say T_1, would lead to a cast insertion that is not type preserving. Indeed, the type system of the cast calculus makes use of the rule above and would be used with e_1 of type $T_1 \rightarrow T_2$, e_2 of type $T'_1 \rightarrow T_3$, and e_3 cast to T_1, and therefore would not match T'_1. Alternatively, one could require the two output consumers to be equal i.e. $T_1 = T'_1$. However, this does not satisfy the criteria of monotonicity with respect to precision. For example, the program $(\mathtt{dapp}\ (\lambda x{:}\star.\,x)\ (\lambda x{:}\mathtt{Int}.\,x)\ 42)$ would not be well-typed but its more precise version $(\mathtt{dapp}\ (\lambda x{:}\mathtt{Int}.\,x)\ 42)\ (\lambda x{:}\mathtt{Int}.\,x)\ 42)$ is well-typed. Our methodology disallows multiple output consumers for the same variable which ensures the uniqueness of the final type.

The next example handles the situation where the final type is a join type. Consider the *case* operator as transformed by this step.

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : T_{11}{}^{\mathrm{p}} + T_{12}{}^{\mathrm{p}} \\ \Gamma, x : T_{11}{}^{\mathrm{c}} \vdash e_2 : T^{\mathrm{p}} \qquad \Gamma, y : T_{12}{}^{\mathrm{c}} \vdash e_3 : T^{\mathrm{p}}\end{array}}{\Gamma \vdash (\mathtt{case}\, e_1\, \mathtt{of}\, \mathtt{inl}\, x \Rightarrow\ e_2 \,|\, \mathtt{inr}\, y \Rightarrow\ e_3) : T^{\mathrm{c}}}$$

$$\Downarrow$$

$$\frac{\begin{array}{c}\Gamma \vdash_G e_1 : T_1 \qquad\qquad T_1 \rhd T_{11} + T_{12} \\ \Gamma, x : T_{11} \vdash_G e_2 : T \qquad \Gamma, x : T_{12} \vdash_G e_3 : T' \\ T^{\mathrm{J}} = T \sqcup T' \qquad T \leadsto T^{\mathrm{J}} \qquad T' \leadsto T^{\mathrm{J}}\end{array}}{\Gamma \vdash_G (\mathtt{case}\, e_1\, \mathtt{of}\, \mathtt{inl}\, x \Rightarrow\ e_2 \,|\, \mathtt{inr}\, y \Rightarrow\ e_3) : T^{\mathrm{J}}}$$

In this example, the final type is the join of T and T', i.e., $T^{\mathrm{J}} = T \sqcup T'$. Recall that the join computes the least upper bound w.r.t. the precision relation, e.g., $(\mathtt{Int}{\rightarrow}\star) \sqcup (\star{\rightarrow}\mathtt{Int}) = \mathtt{Int}{\rightarrow}\mathtt{Int}$. Also, we replace the input T^{c} with T^{J}. Therefore the type of the whole `case` statement is T^{J}.

An interesting example is when a rule contains both a type annotation and an output consumer for the same variable, such as in the rule for an annotated `fix` operator.

$$\frac{\Gamma \vdash e : T^{\mathrm{c}} {\rightarrow} T^{\mathrm{p}}}{\Gamma \vdash (\mathtt{fix}[T]\, e) : T^{\mathrm{c}}} \implies \frac{\begin{array}{c}\Gamma \vdash e : T_1 \\ T_1 \rhd (T' {\rightarrow} T'') \\ T'' \leadsto T \qquad T \leadsto T'\end{array}}{\Gamma \vdash (\mathtt{fix}[T]\, e) : T}$$

Since the type annotation T is present, it is chosen as the final type. We then connect the producer T'' to the final type with the flow $T'' \leadsto T$. Next, we connect the final type T to the consumer T', yielding the flow $T \leadsto T'$. The reason why the type annotation flows into the consumer might seem arbitrary, however the reader should notice how the flows $T \leadsto T'$ and $T'' \leadsto T$ naturally induce the higher order flow $T' {\rightarrow} T'' \leadsto T {\rightarrow} T$ because of the contravariance of the arrow type.

The flow information that we inserted in this step is used to guide the use of consistency in Section 3.6 and the insertion of casts in Section 3.7.

3.5 Restrict Lone Inputs to Be Static

A type system can have variables that appear in input position only, so they do not receive a value from an output. We call these variables *lone inputs*. The correct treatment for lone inputs stems from the recent work of Garcia and Cimini [10]. The idea is to make sure that these variables range over static types only. We define the predicate *static*(T) to hold when $\star$ does not occur in T.

Require lone input variables to satisfy the static predicate.

An example of this situation is the typing rule for abstraction in the implicitly typed λ-calculus.

$$\frac{\Gamma, x : T_1{}^{\mathrm{c}} \vdash e : T_2{}^{\mathrm{p}}}{\Gamma \vdash \lambda x.e : T_1{}^{\mathrm{c}} \rightarrow T_2{}^{\mathrm{c}}} \implies \frac{\begin{array}{c}\Gamma, x : T_1 \vdash_G e : T_2 \\ \mathit{static}(T_1)\end{array}}{\Gamma \vdash_G \lambda x.\, e : T_1 \rightarrow T_2}$$

If we omitted the *static* requirement, T_1 could range over gradual types including $\star$ and the program $(\lambda x.\, x\ x)$ would be well-typed. However, this program is not well-typed in the original language, so the conservativity criteria would not be satisfied [24].

3.6 Replace Flow With Consistency

The previous steps produce gradual type systems and cast insertion procedures that make use of internal information such as flow premises. These premises are precious for driving the methodology towards the correct systems but they can be dismissed/replaced for the final result of the methodology. We apply the following step.

Remove flows to join types and replace the remaining flow premises with consistency premises.

Let us consider the following example.

$$\frac{\Gamma \vdash e : T_1 \qquad T_1 \rhd \mathtt{list}\, T' \qquad T' \leadsto T}{\Gamma \vdash \mathtt{head}[T]\ e : T}$$

$$\Downarrow$$

$$\frac{\Gamma \vdash e : T_1 \qquad T_1 \rhd \mathtt{list}\, T' \qquad T' \sim T}{\Gamma \vdash \mathtt{head}[T]\ e : T}$$

Because $\leadsto$ and $\sim$ denote the same relation, the rules are equivalent.

The next example applies the flow removal step to the result of the cast insertion procedure for the if-then-else operator. Here we remove the highlighted flows to the join type.

$$\frac{\begin{array}{cc} \Gamma \vdash\ e_1 \rightsquigarrow e_1' : T_1 & T_1 \rhd \texttt{Bool} \\ \Gamma \vdash\ e_2 \rightsquigarrow e_2' : T & \Gamma \vdash\ e_3 \rightsquigarrow e_3' : T' \\ T \sqcup T' = T^{\text{J}} \quad \boxed{T \rightsquigarrow T^{\text{J}}} & \boxed{T' \rightsquigarrow T^{\text{J}}} \end{array}}{\begin{array}{l} \Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \rightsquigarrow \\ \left(\begin{array}{l} \texttt{if}\ (e_1 : T_1 \Rightarrow^{l_1} \texttt{Bool})\ \texttt{then}\ (e_2 : T \Rightarrow^{l_2} T^{\text{J}}) \\ \texttt{else}\ (e_3 : T' \Rightarrow^{l_3} T^{\text{J}}) \end{array} \right) : T^{\text{J}} \end{array}}$$

Because $T \sqcup T' = T^{\text{J}}$ implies $T' \sim T^{\text{J}}$ and $T'' \sim T^{\text{J}}$, the resulting rule is equivalent.

3.7 Compilation to the Cast Calculus

Deriving the cast insertion procedure begins with the result from step 5 of the methodology (Section 3.5), so the flow premises are still present in the rules. In this step we create casts based on both the flow premises and pattern matching premises. While the former leads to the obvious cast, the treatment for the latter is more involved. Consider for instance the two pattern matching premises $T \rhd (T_1 \to T_2)$ and $T_2 \rhd (T_{21} \times T_{22})$. In this case, a subexpression of type T should be cast to $T_1 {\to} (T_{21} \times T_{22})$. Moreover, if we additionally had a flow premise $T_{21} \rightsquigarrow T_{21}'$ then the cast should be to $T_1 \to (T_{21}' \times T_{22})$ (as $\times$ is covariant in its arguments). If we had instead a flow premise $T_1' \rightsquigarrow T_1$ then the cast should be to $T_1' \to (T_{21} \times T_{22})$ because of the contravariance of the function type. The cast induced by a pattern-matching variable is therefore obtained by expanding the nested pattern matching variables and by replacing variables according to the flow premises and according to the covariance/contravariance of type constructors. We call this resulting type the *cast destination* for a pattern matching variable[2]. We apply the following step.

For each translated subexpression $e' : T$,

1. *If there is a flow $T \rightsquigarrow T'$, then wrap e' in the cast $e' : T \Rightarrow T'$.*
2. *If there is a pattern matching premise $T \rhd T'$, then wrap e' in the cast $e' : T \Rightarrow T''$ where T'' is the cast destination of T.*

As an example, consider cast insertion for the *cons* operator.

$$\frac{\begin{array}{ccc} \Gamma \vdash e_1 : T' & \Gamma \vdash e_2 : T_2 & T_2 \rhd \texttt{list}\, T'' \\ T' \rightsquigarrow T & & T'' \rightsquigarrow T \end{array}}{\Gamma \vdash \texttt{cons}[T]\, e_1\, e_2 : \texttt{list}\, T}$$
$$\Downarrow$$
$$\frac{\begin{array}{ccc} \Gamma \vdash e_1 \rightsquigarrow e_1' : T' & \Gamma \vdash e_2 \rightsquigarrow e_2' : T_2 & T_2 \rhd \texttt{list}\, T'' \\ T' \rightsquigarrow T & & T'' \rightsquigarrow T \end{array}}{\begin{array}{l} \Gamma \vdash \texttt{cons}[T]\, e_1\, e_2 \\ \quad \rightsquigarrow \texttt{cons}[T]\ \boxed{e_1' : T' \Rightarrow^{\ell}_1 T}\ \boxed{e_2' : T_2 \Rightarrow^{\ell}_2 \texttt{list}\, T} \\ \quad : \texttt{list}\, T' \end{array}}$$

Because of the flow $T' \rightsquigarrow T$, we generate the cast $e_1' : T' \Rightarrow^{\ell_1} T$. For e_2', we have matching $T_2 \rhd \texttt{list}\, T''$ and a flow $T'' \rightsquigarrow T$, so we generate the cast $e_2' : T_2 \Rightarrow^{\ell_2} \texttt{list}\, T$. Notice that the entire generated expression is well-typed by the original rule for `cons`, and therefore well-typed by the type system of the cast calculus.

Next, consider again the `fix` example. It differs from the above `cons` example in that it involves a function type which is contravariant in its first argument. The following is the derived cast insertion rule for (type annotated) `fix`.

$$\frac{\begin{array}{cc} \Gamma \vdash\ e \rightsquigarrow e' : T_1 & T_1 \rhd (T' \to T'') \\ T'' \rightsquigarrow T & T \rightsquigarrow T' \end{array}}{\Gamma \vdash\ \texttt{fix}[T]e \rightsquigarrow \texttt{fix}[T]\ \boxed{e' : T_1 \Rightarrow^{l}_1 T {\to} T}\ : T}$$

Since we have the subexpression e' at type T_1, pattern matching $T_1 \rhd (T' {\to} T'')$ and flows $T'' \rightsquigarrow T$ and $T \rightsquigarrow T'$, we cast the translated subexpression e' from T_1 to $T {\to} T$ (thanks to the contravariance of the function type). Note that the generated `fix` expression is well typed.

4. Gradualizing Type Systems

We have applied our methodology to a variety of type systems. Namely, we have apply the methodology to the STLC, unit types, pairs, tuples, let binding, let rec binding, general recursion (fix), sum types, exceptions, references, lists, if-then-else, and STLC with integers and addition. Figure 7 shows a handful of the generated typing rules. The first column shows the input, the second column shows the gradual type system generated by our methodology, and the third column shows the translated term of the cast insertion (the premises and the output type are the same as for the gradual type system in the second column). The gradual systems for exceptions, lists, and sum types of Figure 7 are novel.

5. Type Systems as Logic Programs

We now proceed to develop an automatic procedure for manipulating type systems according to our methodology. Here we model type systems as logic programs in the intuitionistic theory of higher-order hereditary Harrop formulas [18]. Our implementation (Section 8) works on type systems expressed in λ-prolog, a concrete implementation of this logic. We make use of several features of the logic that suits our needs: types, higher order abstract syntax, and hypothetical reasoning.

Typed Logic Programming Logic programs are equipped with a signature, ranged over by the symbol Σ. The signature defines the entities that are involved in the program. The following is the signature for the STLC.

$$\begin{array}{l} \mathit{term} : \mathit{kind} \\ \mathit{type} : \mathit{kind} \\ \mathit{arrow} : \mathit{type} \to \mathit{type} \to \mathit{type} \\ \mathit{typeof}\ : \mathit{term} \to \mathit{type} \to o \\ \mathit{abs} : \mathit{type} \to (\mathit{term} \to \mathit{term}) \to \mathit{term} \\ \mathit{app} : \mathit{term} \to \mathit{term} \to \mathit{term} \end{array}$$

The kind for propositions is written o. Thanks to these declarations, expressions such as $\mathit{typeof}\ \ T\ T$, for a logical variable T, and $(\mathit{app}\ \mathit{arrow}\ \mathit{arrow})$ are not well-typed.

After the signature, a logic program contains a set of rules. To simplify the Gradualizer, we shall restrict the form of rules as defined below. (t ranges over logic terms, which we define next.)

Definition 1 (Formulae, premises and rules over a signature Σ)**.**

$$\begin{array}{lcl} \mathit{formula} & ::= & \mathit{pred}\, t\ \ldots\ t \\ \mathit{premise} & ::= & \mathit{formula} \mid (\forall x.\mathit{formula} \Rightarrow\ \mathit{formula}) \end{array}$$

A rule *is of the form*

$$\frac{\mathit{premise}_1 \ldots \mathit{premise}_n}{\mathit{formula}}$$

where pred is a predicate name from the signature Σ.

The *terms* of higher-order logic programs consist of λ-terms, logic variables (X), and applications of constructors (f) from the signature.

[2] In Section 6.6 we give a unified definition of cast destination for both flow and pattern matching premises (Definition 14).

Lists

$$\frac{\Gamma \vdash\ e : \texttt{list}\, T^{\,\mathtt{p}}}{\Gamma \vdash\ \texttt{head}[T] : T^{\,\mathtt{c}}} \qquad \frac{\Gamma \vdash\ e : T_1 \quad T_1 \rhd \texttt{list}\, T' \quad T' \sim T}{\Gamma \vdash\ \texttt{head}[T]\; e : T} \qquad \rightsquigarrow \texttt{head}[T]\; (e' : T_1 \Rightarrow^{l_1} \texttt{list}\, T)$$

$$\frac{\begin{array}{c}\Gamma \vdash\ e_1 : T^{\,\mathtt{p}} \\ \Gamma \vdash\ e_2 : \texttt{list}\, T^{\,\mathtt{p}}\end{array}}{\Gamma \vdash\ \texttt{cons}[T]\, e_1\, e_2 : \texttt{list}\, T^{\,\mathtt{c}}} \qquad \frac{\begin{array}{c}\Gamma \vdash\ e_1 : T' \\ \Gamma \vdash\ e_2 : T_1 \quad T_1 \rhd \texttt{list}\, T'' \\ T' \sim T \qquad T'' \sim T\end{array}}{\Gamma \vdash\ \texttt{cons}[T]\, e_1\, e_2 : \texttt{list}\, T} \qquad \begin{array}{l}\rightsquigarrow \texttt{cons}[T]\; (e_1' : T' \Rightarrow^{l_1} T) \\ \qquad\qquad (e_2' : T_1 \Rightarrow^{l_2} \texttt{list}\, T)\end{array}$$

Exceptions

$$\frac{\begin{array}{c}\Gamma \vdash\ e_1 : T^{\,\mathtt{p}} \\ \Gamma \vdash\ e_2 : \texttt{ExcType} \to T^{\,\mathtt{p}}\end{array}}{\Gamma \vdash \texttt{try}\, e_1\, \texttt{with}\, e_2 : T^{\,\mathtt{c}}} \qquad \frac{\begin{array}{c}\Gamma \vdash\ e_1 : T \qquad \Gamma \vdash\ e_2 : T_1 \\ T_1 \rhd T_2 \to T' \quad T_2 \rhd \texttt{ExcType} \\ T \sqcup T' = T^{\mathtt{J}}\end{array}}{\Gamma \vdash \texttt{try}\, e_1\, \texttt{with}\, e_2 : T^{\mathtt{J}}} \qquad \begin{array}{l}\rightsquigarrow \texttt{try}\; (e_1' : T \Rightarrow^{l_1} T^{\mathtt{J}})\; \texttt{with} \\ \qquad (e_2' : T_1 \Rightarrow^{l_2} \texttt{ExcType} \to T^{\mathtt{J}})\end{array}$$

References

$$\frac{\Gamma \vdash e : \texttt{Ref}\ T^{\,\mathtt{pc}}}{\Gamma \vdash\ !e : T^{\,\mathtt{c}}} \qquad \frac{\Gamma \vdash e : T_1 \quad T_1 \rhd \texttt{Ref}\, T}{\Gamma \vdash\ !e : T} \qquad \rightsquigarrow\ !(e' : T_1 \Rightarrow^{l_1} \texttt{Ref}\, T)$$

$$\frac{\begin{array}{c}\Gamma \vdash\ e_1 : \texttt{Ref}\ T^{\,\mathtt{pc}} \\ \vdash\ e_2 : T^{\,\mathtt{p}}\end{array}}{\Gamma \vdash\ e_1 := e_2 : \texttt{unit}^{\,\mathtt{c}}} \qquad \frac{\begin{array}{c}\Gamma \vdash\ e_1 : T_1 \quad T_1 \rhd \texttt{Ref}\, T \\ \vdash\ e_2 : T' \qquad T' \sim T\end{array}}{\Gamma \vdash\ e_1 := e_2 : \texttt{unit}} \qquad \rightsquigarrow (e_1' : T_1 \Rightarrow^{l_1} \texttt{Ref}\ T) := (e_2' : T' \Rightarrow^{l_2} T)$$

General recursion

$$\frac{\Gamma \vdash\ e : (T^{\,\mathtt{c}} \to T^{\,\mathtt{p}})}{\Gamma \vdash (\texttt{fix}\, e) : T^{\,\mathtt{c}}} \qquad \frac{\Gamma \vdash\ e : T_1 \quad T_1 \rhd T \to T' \quad T' \sim T}{\Gamma \vdash (\texttt{fix}\, e) : T} \qquad \rightsquigarrow \texttt{fix}\, (e' : T_1 \Rightarrow^{l_1} T \to T)$$

Let rec (with type annotation)

$$\frac{\begin{array}{c}\Gamma, x : T_1^{\,\mathtt{c}} \vdash\ e_1 : T_1^{\,\mathtt{p}} \\ \Gamma, x : T_1^{\,\mathtt{c}} \vdash\ e_2 : T_2^{\,\mathtt{p}}\end{array}}{\Gamma \vdash (\texttt{letrec}\, x : T_1 = e_1\, \texttt{in}\, e_2) : T_2^{\,\mathtt{c}}} \qquad \frac{\begin{array}{c}\Gamma, x : T_1 \vdash\ e_1 : T_1' \\ \Gamma, x : T_1 \vdash\ e_2 : T_2 \quad T_1' \sim T_1\end{array}}{\Gamma \vdash (\texttt{letrec}\, x : T_1 = e_1\, \texttt{in}\, e_2) : T_2} \qquad \begin{array}{l}\rightsquigarrow \texttt{letrec}\, x : T_1' = (e_1' : T_1' \Rightarrow^{l_1} T_1) \\ \quad \texttt{in}\, e_2'\end{array}$$

If-then-else

$$\frac{\begin{array}{c}\Gamma \vdash\ e_1 : \texttt{Bool}^{\,\mathtt{p}} \\ \Gamma \vdash\ e_2 : T^{\,\mathtt{p}} \quad \Gamma \vdash\ e_3 : T^{\,\mathtt{p}}\end{array}}{\Gamma \vdash (\texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3) : T^{\,\mathtt{c}}} \qquad \frac{\begin{array}{c}\Gamma \vdash\ e_1 : T_1 \quad T_1 \rhd \texttt{Bool} \\ \Gamma \vdash\ e_2 : T \quad \Gamma \vdash\ e_3 : T' \\ T \sqcup T' = T^{\mathtt{J}}\end{array}}{\Gamma \vdash (\texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3) : T^{\mathtt{J}}} \qquad \begin{array}{l}\rightsquigarrow \texttt{if}\ \ (e_1' : T_1 \Rightarrow^{l_1} \texttt{Bool}) \\ \quad \texttt{then}\, (e_2' : T \Rightarrow^{l_2} T^{\mathtt{J}}) \\ \quad \texttt{else}\, (e_3' : T' \Rightarrow^{l_3} T^{\mathtt{J}})\end{array}$$

Figure 7. Example applications of the methodology. The left-hand column contains the static typing rules (the input), the middle column gives the gradual typing rules (output), and the right-hand column excerpts just the conclusion of the cast insertion rules (output). The primed version of each expression is the result of recursively applying cast insertion, i.e., e' is the cast inserted version of e.

Definition 2 (Logic Terms)**.**

$$\textit{term}\ t \quad ::= \quad x \mid \lambda x.t \quad \mid (t\ t) \quad \mid X \quad \mid \quad (f\ t\ \ldots\ t)$$

For the sake of clarity, we sometimes use E, T, and R instead of X, for denoting variables of kind *term* (E), kind *type* (T) and abstractions (R). Given a rule r, premises(r) denotes its set of premises, conclusion(r) denotes the formula in its conclusion and vars(r) denotes the set of logic variables that occur in r and similarly for vars(t).

Higher Order Abstract Syntax Our type systems expressed as logic programs make use of higher order abstract syntax (HOAS). As such, we can use the λ abstraction in the logic to represent variable binding in the object language. For example, in the above signature for the STLC, we declared the constructor for STLC abstraction (*abs*) as taking just two parameters: the type annotation and a logic abstraction from terms to terms. For example, the following term of the STLC

$$(\lambda f{:}\texttt{Bool}{\to}\texttt{Bool}.\ \lambda x{:}\texttt{Bool}.\ f\ x)$$

is encoded as follows

$$(\textit{abs}\ (\textit{arrow}\ \texttt{Bool}\ \texttt{Bool})\ (\lambda f.\ (\textit{abs}\ \texttt{Bool}\ (\lambda x.\ (\textit{app}\ f\ x)))))$$

Hypothetical Reasoning To appreciate the role of hypothetical reasoning, let us consider how we can define the typing rule for abstraction of STLC in this setting.

$$\frac{(\forall x.\textit{typeof}\ x\, T_1\ \Rightarrow\ \textit{typeof}\ (R\ x)\, T_2)}{\textit{typeof}\ (\textit{abs}\ T_1\ R)\ (\textit{arrow}\ T_1\ T_2)}$$

An environment Γ is not necessary because we instead use a combination of universal quantification and implication provided by the logic. Operationally speaking, encountering the subgoal

$$(\forall x.\textit{typeof}\ x\, T_1\ \Rightarrow\ \textit{typeof}\ (R\ x)\, T_2)$$

creates a fresh constant for x and temporarily augment the logic program with the fact *typeof* $x\, T_1$ while trying to prove the goal *typeof* $(R\ x)\, T_2$.

We now have all the ingredients for defining type systems as logic programs. Type systems are simply logic programs whose signature contains the kinds *term* and *type* and also a distinguished typeability predicate.

Definition 3 (Type System)**.** *A type system is a triple* (Σ, D, p) *where* Σ *is a signature,* D *is a set of rules over* Σ*, and* p *is a* *distinguished declaration in Σ for typeability. Also $term : kind \in \Sigma$, $type : kind \in \Sigma$, and $p : term \to type \to o \in \Sigma$.*

6. The Gradualizer

In this section, we present an algorithm for turning type systems into their gradually typed version and for producing the compilation to the cast calculus.

The gradualization of a type system is the composition of steps that we define in this section:

$$(\mathbb{T} \xrightarrow{\texttt{toGr}} \mathbb{T}^C) = \mathbb{T} \xrightarrow{\texttt{toPM}} \mathbb{T}^p \xrightarrow{\texttt{toFlow}} \mathbb{T}^f \xrightarrow{\texttt{toSt}} \mathbb{T}^G \xrightarrow{\texttt{toCnst}} \mathbb{T}^C$$

The generation of the compiler to the cast calculus shares many of the same steps, but also include the cast insertion step $\xrightarrow{\texttt{toCI}}$.

$$(\mathbb{T} \xrightarrow{\texttt{toComp}} \mathbb{T}^{CC}) = \mathbb{T} \xrightarrow{\texttt{toPM}} \mathbb{T}^p \xrightarrow{\texttt{toFlow}} \mathbb{T}^f \xrightarrow{\texttt{toSt}} \mathbb{T}^G \xrightarrow{\texttt{toCI}} \mathbb{T}^c \xrightarrow{\texttt{toCnst}} \mathbb{T}^{CC}$$

These steps make use of several auxiliary functions. The function $\mathrm{sig}(pred, k)$ returns the kind of the k-th argument of $pred$, for example $\mathrm{sig}(\mathit{typeof}, 1) = term$ and $\mathrm{sig}(\mathit{typeof}, 2) = type$. The function $\mathrm{lone}(X, r)$ is true whenever the variable X only appears in r in input positions.

6.1 Step 1 and 2: Input/Output and Producer/Consumer

The user of the Gradualizer provides a function $\mathrm{mode}(pred, k) = m$, where $m \in \{in, out\}$, that says whether the k-th argument of the relation $pred$ is an input or an output. As an example, $\mathrm{mode}(\mathit{typeof}, 1) = in$ and $\mathrm{mode}(\mathit{typeof}, 2) = out$. In our notion of type systems, we have a convenient way for detecting whether a particular variable is in input or output position. For example, type environments are inputs and in our setting their information is encoded as a hypothetical appeal to the same predicate *typeof*. Thanks to this, we can use mode applied to *typeof* with swapped information (inputs become outputs and vice versa) when we inspect the left side of an implication w.r.t. the premises of a rule. Similarly, the conclusion of a rule experiences the same swap. This time, the conclusion is on the right side of an implication (the left side contains the set of premises). To summarize, we have a uniform way to detect our input/output information simply by considering the swapped information from mode any time we cross an implication (either to the left or to the right). To account for this, we use the notation f^{-1} to refer to an altered version of the function f where all occurrences of in and out in the definition of f have been flipped.

The user also provides a function that specifies the polarity of type constructors. The function $\mathrm{contra}(f, k)$ should be true if the k-th argument of the type constructor f is contravariant or simultaneously covariant and contravariant. As an example, $\mathrm{contra}(\texttt{arrow}, 1) = true$. Because contravariance and covariance swap when traversing a contravariance argument, we shall use this function negated and write $\neg\mathrm{contra}(f, k)$.

The $\mathrm{usertype}(X, r)$, $\mathrm{consumer}(X, r)$ and $\mathrm{producer}(X, r)$ functions are true whenever X appears in the rule r as a type annotation, consumer, or producer, respectively as described in Section 3.2.

6.2 Restrictions of the Gradualizer

The definition of type systems above is very liberal and includes a broad class of logic programs. However, many logic programs do not make sense as type systems. Moreover, as we are set to provide an algorithm and reason about it, we would like the input language to be simple. This avoids introducing machinery for corner cases or capturing unnecessary generality.

We should however restrict to a meaningful fragment of type systems that is still expressive enough for all our envisioned use cases. To this aim, we define a notion of *well-formed type system.*

Definition 4 (Well-formed type system)**.** *Given a type system $\mathbb{T} = (\Sigma, D, \mathit{typeof})$ we say that $\mathbb{T}$ is well-formed whenever*

1. *Every rule of $\mathbb{T}$ is well-moded, i.e. the input/output dependency relation of variables is* acyclic.
2. *The conclusion of every rule is of the form $\mathit{typeof}\ (f\ X_1 \ldots X_k)\ t$, ($t$ might be a constructed type) where $X_1 \ldots X_k$ are distinct variables.*
3. *The form of premises is $\mathit{typeof}\ E\ t$ or $(\forall x.\mathit{typeof}\ x\ X \Rightarrow \mathit{typeof}\ (R\ x)\ t)$, where variables E and R are used only once in premises and they appear in the conclusion.*
4. *Restrictions to HOAS: Abstractions have kind $(term \to term)$ only and the only usage of HOAS in the syntax is with $(R\ x)$ of Restriction 3.*
5. *Uniqueness of final type: a variable does not appear more than once in output consumer position.*
6. *Restriction to simple types: Declarations for the kind type are of the form: $f : \underbrace{type \to \ldots \to type}_{n\ times} \to type$ in Σ, with $n \geq 0$.*

Restriction 1 is a common restriction for many reasonable logic programs. In our setting this restriction is crucial for proofs: premises can be ordered and when inspecting them in the context of a proof we can always assume its variables have been previously instantiated (preserving inductive properties) or are lone. Restriction 2 and Restriction 3 prescribe a shape of conclusions and premises that facilitates detecting which variables need to be cast. Restriction 2 imposes the use of distinct variables to ensure the monotonicity criteria. Restriction 3 disallows a variable to be typed more than once because that would introduce difficulties in determining which cast to apply. Restriction 4 is a conservative choice to make the procedure simpler. In principle, more complicated abstractions and nested implications could be captured. As discussed in Section 3.4, Restriction 5 is necessary for avoiding ambiguity in the flow for types. Restriction 6 confines the scope of the Gradualizer to the family of simply typed systems. Addressing more sophisticated types is future work.

Examples of Well-Formed Type Systems Restrictions of Definition 4 are liberal enough for admitting interesting type systems. The type systems of Figure 7 and those mentioned in Section 4 are all well-formed. As a concrete example of a typing rule as a logic rule, we show the typing rule for *case* on disjoint sums:

$$\text{(case)}\ \frac{\begin{array}{c}\mathit{typeof}\ \ E\ (T_1 + T_2)\\ (\forall x.\mathit{typeof}\ x\ T_1 \Rightarrow\ \mathit{typeof}\ \ (R_1\ x)\ T)\\ (\forall y.\mathit{typeof}\ y\ T_2 \Rightarrow\ \mathit{typeof}\ \ (R_2\ y)\ T)\end{array}}{\mathit{typeof}\ \ (\mathit{case}\ E\ R_1\ R_2)\ T}$$

We have proved that for well-formed systems, the Gradualizer produces a correct gradual type system and a correct compilation to the cast calculus (Section 7).

Predicates for Gradual Typing The Gradualizer introduces new predicates into the type systems in input. In particular, we use the predicates *flow* (for flow premises), *join* (for the join operator) and *static* (for static premises). As the signature and defining rules for these operators are standard or trivial we do not show their automatic generation. Of course, the implementation of the Gradualizer generates them. Another predicate is pmatch_f, which corresponds to the pattern matching relation $\triangleright$ discussed earlier, but indexed by the type constructor to be matched.

The following subsections describe the Gradualizer through the steps of our methodology.

6.3 Step 3: Pattern Matching of Constructed Outputs

We first generate the pattern matching predicates by inspecting the declarations in the signature. Next, we introduce the step $\overset{\texttt{toPM}}{\Longrightarrow}$ for pattern matching constructed outputs.

Definition 5 (Type Systems with Pattern Matching)**.** *A type system* $\mathbb{T}' = (\Sigma', D', \mathit{typeof})$ extends a type system $\mathbb{T} = (\Sigma, D, \mathit{typeof})$ with pattern matching *whenever* $\Sigma \subseteq \Sigma'$ *and for all declarations* $f : \underbrace{type \to \ldots \to type}_{n\ times} \to type$ *in* Σ*, with* $n \geq 0$*, it holds that*

- Σ' *contains:* $\quad pmatch_f : type \to \underbrace{type \to \ldots \to type}_{n\ times} \to o.$
- D' *contains the rules*

$$\begin{array}{l} pmatch_f\ (f\ X_1\ \ldots\ X_n)\ X_1\ \ldots\ X_n. \\ pmatch_f\ \underbrace{\star\ \star\ \ldots\ \star}_{(n+1)\ times}. \end{array}$$

where X_i *are distinct logic variables for* $1 \leq i \leq n$.

Given a premise $pmatch_f\ X\ X_1\ \ldots\ X_n$ *in a rule* r*, we say that* X is pattern-matched in r.

Definition 6 ($\mathbb{T} \overset{\texttt{toPM}}{\Longrightarrow} \mathbb{T}^p$)**.** *Given type systems* $\mathbb{T} = (\Sigma, D, \mathit{typeof})$ *and* $\mathbb{T}^p = (\Sigma', D', \mathit{typeof})$ *we write* $\mathbb{T} \overset{\texttt{toPM}}{\Longrightarrow} \mathbb{T}^p$ *whenever* $\mathbb{T}^p$ *extends* $\mathbb{T}$ *with pattern matching,* Σ' *contains the declaration* $\star : type$ *and* D' *is the least set such that for all rules* r *in* D*,* D' *contains a rule* r' *such that*

- $\mathrm{conclusion(r')} = \mathrm{conclusion(r)}$*, and*
- $\mathrm{premises(r')}$ *is the least set such that for all premises* Φ *of* r*,* $\mathrm{premises(r')}$ *contains the premise* $\mathrm{pm}(\Phi)$

where pm *is defined as follows:*

$$\begin{array}{l}
\mathrm{pm}((\forall x.\Phi_1 \Rightarrow \Phi_2)) = (\forall x.\Phi_1 \Rightarrow \mathrm{pm}(\Phi_2)) \\
\mathrm{pm}(\mathit{pred}\, t_1\ \ldots\ t_n) = \mathit{pred}\, t_1^*\ \ldots\ t_n^* \\
\quad \textit{where } t_k^* = \begin{cases} \mathrm{pm}(t_k) & \textit{if } \mathrm{mode}(\mathit{pred}, k) = \mathit{out} \\ & \textit{and } \mathrm{sig}(\mathit{pred}, k) = \mathit{type}. \\ t_k & \textit{otherwise.} \end{cases} \\
\mathrm{pm}(f\ t_1\ \ldots\ t_n) = X, \textit{with } X \textit{ fresh in } r', \textit{ and} \\
\qquad \mathrm{premises(r')} \textit{ contains } \mathit{pmatch}_f X\ \mathrm{pm}(t_1)\ \ldots\ \mathrm{pm}(t_n). \\
\mathrm{pm}(t) = t, \textit{ otherwise.}
\end{array}$$

6.4 Step 4: Flow Discovery

In this step we formalize how we mark each producers as flowing to its consumers through their final type. To formalize the notion of *final type*, we use functions $X_r^{\mathtt{J}}$, $X_r^{\mathtt{C}}$ and $X_r^{\mathtt{U}}$ that return a variable fresh in rule r. They are injective and have disjoint codomains. These functions return a dedicated fresh variable for join results, output consumers and type annotations, respectively.

Definition 7 (Final type of variables)**.** *Given a variable* X *and a rule* r*, the* final type of X in r*, written* X_r^{Fin} *is defined as follows, (the clauses below apply in order).*

$$\begin{array}{l}
X_r^{Fin} = X_r^{\mathtt{U}}, \textit{if } \mathrm{usertype}(X, r). \\
X_r^{Fin} = X_r^{\mathtt{C}}, \textit{if } \mathrm{consumer}(X, r). \\
X_r^{Fin} = X_r^{\mathtt{J}}, \textit{if } \mathrm{producer}(X, r). \\
X_r^{Fin} = X, \textit{ otherwise.}
\end{array}$$

We also lift $\bullet^{Fin}$ *to logic terms in the obvious way.*

The step for flow discovery, $\overset{\texttt{toFlow}}{\Longrightarrow}$, is divided into two smaller steps, $\overset{\texttt{toFlow}'}{\Longrightarrow}$ and $\overset{\texttt{toJoin}}{\Longrightarrow}$. The first step $\overset{\texttt{toFlow}'}{\Longrightarrow}$ generates flow premises and the second step $\overset{\texttt{toJoin}}{\Longrightarrow}$ inserts premises for computing the join of the producers.

We first describe $\overset{\texttt{toFlow}'}{\Longrightarrow}$ (Definition 8 below). This step accomplishes four tasks. The first task is to assign new variables to outputs and have output producers flow into their final type. This is handled by the last clause of the auxiliary function FL, which is only invoked in output producer contexts (either an output of a predicate or in a covariant position of a pattern match on a producer).

The second task is to replace consumers by their final type. Consumers in input positions are straightforward to find and deal with, as in the clause of FL for predicates (pred). Dealing with consumers in output position is more complex. Such variables can occur within a pattern match (pmatch). A variable X_k on the right-hand side of a pattern match on variable X is a consumer if either (a) k is a contravariant parameter of the type constructor f and if X is a producer (pattern matched variables occur only once in a rule) or (b) k is in covariant parameter and X is a consumer. In these cases X_k is replaced with $X_{r,k}^{\mathtt{C}}$. If the converse of the above is true, then variable X_k is an output producer and handled by the last clause of FL.

The third task is to replace type annotations by the final type. This is handled in the fourth clause of FL, for an application of a constructor f. If parameter k of f is of kind *type*, then X_k is a type annotation and is replaced by $X_{r,k}^{\mathtt{U}}$.

The fourth and final task is to mark the final type as flowing to any consumers that are present. This is accomplished by the last two lines in the definition for $\overset{\texttt{toFlow}'}{\Longrightarrow}$. If the final type is a variable that appears in an annotation, then we generate a flow from that variable to the consumer. If the final type is the consumer, then there is no need to generate a flow because it would just be to itself.

Definition 8 ($\mathbb{T} \overset{\texttt{toFlow}'}{\Longrightarrow} \mathbb{T}'$)**.** *Given two type systems* $\mathbb{T} = (\Sigma, D, \mathit{typeof})$ *and* $\mathbb{T}' = (\Sigma', D', \mathit{typeof})$ *we write* $\mathbb{T} \overset{\texttt{toFlow}'}{\Longrightarrow} \mathbb{T}'$ *whenever* $\Sigma \subseteq \Sigma'$*,* $\mathbb{T}'$ *defines the predicate flow, and for all rules* r *in* D*,* D' *contains a rule* r' *such that*

- $\mathrm{conclusion(r')} = \mathrm{FL}^{-1}\mathrm{conclusion(r)}$*, and*
- $\mathrm{premises(r')}$ *is the least set such that for all premises* Φ *of* r*,* $\mathrm{premises(r')}$ *contains the premise* $\mathrm{FL}(\Phi)$

where FL *is defined as follows:*

$$\begin{array}{l}
\mathrm{FL}((\forall x.\Phi_1 \Rightarrow \Phi_2)) = (\forall x.\mathrm{FL}^{-1}(\Phi_1) \Rightarrow \mathrm{FL}(\Phi_2)) \\
\mathrm{FL}(\mathit{pmatch}_f\ X\, X_1\ \ldots\ X_n) = \mathit{pmatch}_f\ X_r^{Fin}\ X_1^*\ \ldots\ X_n^* \\
\quad \textit{where } X_k^* = \begin{cases} X_k & \textit{if } X_k \textit{ is pattern-matched in } r, \\ X_{r,k}^{\mathtt{C}} & \textit{if } \mathrm{contra}(f, k) \textit{ and } \mathrm{producer}(X, r), \\ X_{r,k}^{\mathtt{C}} & \textit{if } \neg\mathrm{contra}(f, k) \textit{ and } \mathrm{consumer}(X, r), \\ \mathrm{FL}(X_k) & \textit{otherwise.} \end{cases} \\
\mathrm{FL}(\mathit{pred}\, t_1\ \ldots\ t_n) = \mathit{pred}\, t_1^*\ \ldots\ t_n^* \\
\quad \textit{where } t_k^* = \begin{cases} \mathrm{FL}(t_k) & \textit{if } \mathrm{mode}(\mathit{pred}, k) = \mathit{out}, \\ t_{r,k}^{Fin} & \textit{if } \mathrm{mode}(\mathit{pred}, k) = \mathit{in}. \end{cases} \\
\mathrm{FL}(f\ X_1\ \ldots\ X_n) = (f\ X_1^*\ \ldots\ X_n^*) \\
\quad \textit{where } X_k^* = \begin{cases} X_{r,k}^{\mathtt{U}} & \textit{if } \mathrm{sig}(f, k) = \mathit{type}, \\ X_k & \textit{otherwise.} \end{cases} \\
\mathrm{FL}(X) = X', \textit{for some variable } X' \textit{ fresh in } r', \\
\quad \textit{Also, } \mathrm{premises(r')} \textit{ contains flow } X'\ X_r^{Fin}.
\end{array}$$

Moreover, for all variable $X \in r$*, if* $\mathrm{consumer}(X, r)$ *and also* $\mathrm{usertype}(X, r)$ *then* $\mathrm{premises(r')}$ *contains* $X_r^{\mathtt{U}} \rightsquigarrow X_r^{\mathtt{C}}$.

Next we describe the $\overset{\texttt{toJoin}}{\Longrightarrow}$ step. This step collects all the flow premises whose targets are the join type $X_r^{\mathtt{J}}$ for a variable X and then inserts a premise to compute the join of the producers (sources) of the flows. To specify the collection of flow premises, we define *maximal set of flows*.

Definition 9 (Maximal set of flows)**.** *Given a type system $\mathbb{T}$ and a rule r of $\mathbb{T}$, we say P is a* maximal set of flows for X in r *whenever*

$$P = \{\mathit{flow}\ X_1\, X,\ \mathit{flow}\ X_2\, X, \ldots, \mathit{flow}\ X_k\, X\}$$

for some $k \geq 1$, variables X, X_1, X_2, . . ., X_k, and premises Φ in $\mathrm{premises(r)}$*, either $\Phi \in P$ or $\Phi \neq \mathit{flow}\ X'\, X$, for any X'.*

For a maximal set of flows P, let $\mathrm{source}(P) = \{X_1, \ldots, X_k\}$.

Definition 10 ($\mathbb{T} \overset{\mathtt{toJoin}}{\Longrightarrow} \mathbb{T}^f$)**.** *Given two type systems $\mathbb{T} = (\Sigma, D, \mathit{typeof})$ and $\mathbb{T}^f = (\Sigma', D', \mathit{typeof})$, we say $\mathbb{T} \overset{\mathtt{toJoin}}{\Longrightarrow} \mathbb{T}^f$ whenever $\mathbb{T}^f$ defines the join operator, and for all rules r in D, D′ contains a rule r′ such that*

- $\mathrm{conclusion(r')} = \mathrm{conclusion(r)}$*, and*
- *for all variables X, let P be the maximal set of flows for $X_r^{\mathtt{J}}$. Then, join* $\mathrm{source}(P) = X_r^{\mathtt{J}} \in \mathrm{premises(r')}$.

Flow discovery is the composition of the above two steps:

$$(\mathbb{T} \overset{\mathtt{toFlow}}{\Longrightarrow} \mathbb{T}^f) = \mathbb{T} \overset{\mathtt{toFlow}'}{\Longrightarrow} \mathbb{T}' \overset{\mathtt{toJoin}}{\Longrightarrow} \mathbb{T}^f$$

6.5 Step 5: Staticity for Lone Inputs

Ensuring staticity for lone inputs amounts in spotting lone variables and generating a static premise for them. In this step we also turn *typeof* into typeof_G .

Definition 11 ($\mathbb{T} \overset{\mathtt{toSt}}{\Longrightarrow} \mathbb{T}^G$)**.** *Given type systems $\mathbb{T} = (\Sigma, D, \mathit{typeof})$ and $\mathbb{T}^G = (\Sigma', D', \mathit{typeof}_G)$ we write $\mathbb{T} \overset{\mathtt{toSt}}{\Longrightarrow} \mathbb{T}^G$ whenever $\Sigma[\mathit{typeof}/\mathit{typeof}_G] \subseteq \Sigma'$, $\mathbb{T}^G$ defines the predicate static, and D′ is the least set such that for all rules r in D, D′ contains a rule r′ such that*

- $\mathrm{conclusion(r')} = \mathrm{conclusion(r)}$*, and*
- $\mathrm{premises(r')}$ *is the least set s.t.* $\mathrm{premises(r)} \subseteq \mathrm{premises(r')}$ *and for all $X \in \mathrm{vars}(r)$, if* $\mathrm{lone}(X, r)$ *then* $\mathrm{premises(r')}$ *contains static X.*

Moreover, the predicate typeof is consistently replaced by typeof_G.

6.6 Step 5.5: Compilation to the Cast Calculus

We now tackle the generation of the cast insertion procedure. The first task is to define the type system of the cast calculus. This is a simple extension of the original type system.

Definition 12 (Cast Calculus)**.** *A type system $\mathbb{T} = (\Sigma, D, \mathit{typeof})$ is a* cast calculus *whenever Σ contains the declarations*

$$\begin{array}{l} \mathit{label} : \mathit{kind} \\ \star : \mathit{type} \\ \mathit{compToCC}\ : \mathit{term} \to \mathit{term} \to \mathit{type} \to o \\ \mathit{cast} : \mathit{term} \to \mathit{type} \to \mathit{label} \to \mathit{type} \to \mathit{term} \end{array}$$

and D contains the rule

$$\frac{\mathit{typeof}_{CC}\ \ E\, X_1}{\mathit{typeof}_{CC}\ \ (\mathit{cast}\, E\, X_1\, L\, X_2)\ X_2}$$

Definition 13 (Cast Calculus of a Type System)**.** *Given type systems $\mathbb{T} = (\Sigma, D, \mathit{typeof})$ and $\mathbb{T}^{CC} = (\Sigma', D', \mathit{typeof}_{CC})$, we write $\mathbb{T} \overset{\mathtt{toCC}}{\Longrightarrow} \mathbb{T}^{CC}$ whenever $\Sigma[\mathit{typeof}/\mathit{typeof}_{CC}] \subseteq \Sigma'$, $D[\mathit{typeof}/\mathit{typeof}_{CC}] \subseteq D'$, and $\mathbb{T}^{CC}$ is a cast calculus.*

Before diving into the compilation, we formally define the notion of cast destination we discussed in Section 3.7. To this aim, it is convenient to uniformly treat flow and pattern matching premises.

Definition 14 (Cast destination)**.** *Given a variable X and a rule r, the* cast destination of X in r*, written X_r^{Dest} is defined as follows. (the clauses below apply in order).*

$$\begin{array}{l} X_{r,1}^{\mathit{Dest}} = X_2, \\ \qquad \mathit{if}\, X_2 \rightsquigarrow X_1 \in \mathrm{premises(r)}\ \mathit{and}\ \mathrm{consumer}(X_1, r). \\ X_{r,1}^{\mathit{Dest}} = X_2, \mathit{if}\, X_1 \rightsquigarrow X_2 \in \mathrm{premises(r)}. \\ X_r^{\mathit{Dest}} = (f\ X_{r,1}^{\mathit{Dest}}\ \ldots\ X_{r,n}^{\mathit{Dest}}) \\ \qquad \mathit{if}\, \mathrm{premises(r)}\ \mathit{contains}\ \mathit{pmatch}_f\ X\, X_1\ \ldots\ X_n. \\ X_r^{\mathit{Dest}} = X, \mathit{otherwise}. \end{array}$$

The next step is to dive straight into the compilation. Thanks to Restriction 2 and 3, identifying the terms of the conclusion that are subject to cast is not problematic.

We assume that the function enc_r, that stands for encoding, is injective and that its codomain is disjoint from those of $\bullet_r^{\mathtt{J}}$, $\bullet_r^{\mathtt{C}}$ and $\bullet_r^{\mathtt{U}}$. enc_r is a function from variables of kind `term` to variables that are fresh in the rule r and is the identity on all other variables. Intuitively, enc_r has the role of turning the name e into e' for cast insertion. We also lift enc_r to terms in the obvious way.

As we make use of HOAS, the Gradualizer can encounter logical variables R that represent abstractions (but not explicit λs, thanks to Restriction 3). In that case a cast $R : T_1 \Rightarrow^l T_2$ is not well-typed, therefore our procedure generates a wrapped term $\lambda x.((R\ x) : T_1 \Rightarrow^l T_2)$.

Definition 15 (Cast Calculus with Compilation)**.** *Given type systems $\mathbb{T} = (\Sigma, D, \mathit{typeof}_G)$ and $\mathbb{T}^c = (\Sigma', D', \mathit{typeof}_G)$, we write $\mathbb{T} \overset{\mathtt{toCI}}{\Longrightarrow} \mathbb{T}^c$ whenever $\mathbb{T}^c$ is a cast calculus and for all rules r in D that define typeof_G, D′ contains a rule r′ such that*

- $\mathrm{conclusion(r')} = \mathrm{cast}^{-1}(\mathrm{conclusion(r)})$*, and*
- $\mathrm{premises(r')}$ *is the least set such that for all premises Φ of r,* $\mathrm{premises(r')}$ *contains the premise* $\mathrm{cast}(\Phi)$.

where cast *is defined as follows:*

$$\begin{array}{l} \mathrm{cast}((\forall x.\Phi_1 \Rightarrow \Phi_2)) = (\forall x.\mathrm{cast}(\Phi_1)) \Rightarrow \mathrm{cast}(\Phi_2) \\ \mathrm{cast}(\mathit{typeof}_G\, e\, t) = (\mathit{compToCC}\ \ e\, e^*\ t) \\ \mathit{where}\ e^* = \begin{cases} \mathrm{cast}(e) & \mathit{if}\, \mathrm{mode}(\mathit{typeof}_G, 1) = \mathit{out}. \\ \mathrm{enc}_r(e) & \mathit{otherwise.} \end{cases} \\ \mathrm{cast}(\Phi) = \Phi, \mathit{otherwise.} \\ \mathrm{cast}(e) = e\sigma, \end{array}$$

where σ is defined as: for all variables E (or R) $\in \mathrm{vars}(e)$,

- *if $\mathit{typeof}_G\ E\ X \in r$ then $\sigma(E) = \mathit{cast}\ (\mathrm{enc}_r(E))\ X\ L\ X_r^{\mathit{Dest}}$.*
- *if $\Phi \Rightarrow \mathit{typeof}_G\ (R\ x)\ X \in r$, then $\sigma(R) =$ $\lambda x.\ \mathit{cast}\ (\mathrm{enc}_r(R\ x))\ X\ L\ X_r^{\mathit{Dest}}$.*

The substitution σ is simply enc_r *everywhere else. In each case, L is a fresh variable in r. To avoid unnecessary casts, casts $X \Rightarrow^L X_r^{\mathit{Dest}}$ are omitted when $X = X_r^{\mathit{Dest}}$.*

6.7 Step 6: Replace Flow with Consistency

This step, written $\overset{\mathtt{toCnst}}{\Longrightarrow}$, is straightforward so we omit the formal definition. This step merely removes flows to join types, such as *flow* $X\, X_r^{\mathtt{J}}$, and replaces the remaining flows in a rule with consistency checks, that is, each premise of the form *flow* $T\, T'$ is replaced with *consistent* $T\, T'$. See Section 3.6 for examples.

7. Correctness of the Gradualizer

We have proved that given a type system $\mathbb{T}$, if $\mathbb{T}$ is well-formed, $\mathbb{T} \overset{\mathtt{toGr}}{\Longrightarrow} \mathbb{T}^G$ and $\mathbb{T} \overset{\mathtt{toComp}}{\Longrightarrow} \mathbb{T}^{CC}$ then the resulting type system and compiler satisfy the correctness criteria described in Section 2. We summarize the proofs below.

Conservative Extension The idea of the proof is that, over static programs and types, the pattern matching premises collapse to perform the ordinary pattern-matching (the clause for $\star$ is never used), flow premises collapse to equality checking and join computations holds only for equal types, i.e. $T = T_1 \sqcup T_2$ iff $T_1 = T_2 = T$.

Monotonicity w.r.t. Precision The proof mainly relies on the fact that pattern-matching and join premises give less precise outputs at less precise inputs. For instance, $\texttt{Int} \rightarrow \texttt{Int} \triangleright \texttt{Int} \rightarrow \texttt{Int}$ and $\texttt{Int} \rightarrow \star \triangleright \texttt{Int} \rightarrow \star$ with indeed $\texttt{Int} \sqsubseteq \texttt{Int}$ and $\star \sqsubseteq \texttt{Int}$. Also, flow premises (consistency) continue to hold on less precise inputs.

Type Preservation of Cast Insertion The first insight is that since typeof_G and $\mathit{compToCC}$ contains the same premises, these are satisfied for typeof_G as much as they are for $\mathit{compToCC}$ (and with same instantiations). What connects the dots is however the following. typeof_{CC} makes use of the original rule in typeof and therefore more program variables might be typed at a same type T. In $\mathit{compToCC}$ those programs are typed at different types but the encoding is instrumented so to have them cast to a same type T^{Fin} (the final type for them). Therefore, with in mind the typing rule for the cast operator, typeof_{CC} can type the corresponding expression at a type that makes use of T^{Fin} in place of T (if it appears in the assigned type). This however syncs with the same type assigned by both typeof_G and $\mathit{compToCC}$ (remember, they coincide w.r.t. this matter). Indeed, the occurrences of T in the original typing rule typeof (and so in typeof_{CC}) are inputs that in typeof_G have been replaced by T^{Fin}.

Monotonicity of Cast Insertion This proof relies on insights that are similar to those of the proof of the monotonicity of typeof_G w.r.t. the precision relation.

8. The Implementation of the Gradualizer

We have implemented the Gradualizer in Haskell. This tool takes in input the implementation of a type system in λ-prolog and produces the type checker and the compilation procedure to the cast calculus in λ-prolog. Currently, language designers can specify the covariance/contravariance of type constructors with a special commented tag in the logic program in input (details can be found in the link below). We have applied our tool to the type systems mentioned in Section 4 and the generated type checkers and compilers are part of the tool. The Gradualizer can be found at the following github repository:

`https://github.com/mcimini/Gradualizer`.

9. Future Work

We discuss several directions for future work on the Gradualizer.

Extension to Richer Type Systems The Gradualizer handles a simple typing judgment of the form $\Gamma \vdash e : T$. In principle, the work in this paper can be adapted to other typing judgments. For example, extending the methodology to handle bidirectional type checking does not seem problematic.

The Gradualizer currently captures a number of type systems that are confined within the family of simply typed systems. It remains to be explored whether we can extend the methodology to more sophisticated type systems such as recursive types, universal types [3], or even dependent types.

Auxiliary Predicates, e.g., Subtyping We have explored extending the Gradualizer to subtyping and it does not seem difficult. Going further, we shall investigate a methodology for any *auxiliary predicate* that the language designer might use. The recent work of Garcia et al. [11] provide guidance for how to port arbitrary relations to the gradual typing world, which we plan to use.

Dynamic Semantics In this paper, we have shown how to derive the static aspects of gradual typing. Deriving the dynamic aspects is an important area of future work. Our goal will be to provide a methodology for deriving the operational semantics of the cast calculus from rules that define the operational semantics of the original language. Of particular importance, we are planning to study space efficient casts [9, 13, 14, 25] and the derivation of correct blame tracking reductions [26, 37].

10. Conclusions

In this paper, we have described a methodology for gradual typing and validated it through formal results on transformations of type systems. This paper is meant to serve as a reference for language designers who want to augment their languages with gradual typing. In this regard, we believe that our methodology, algorithm, and implementation will be essential tools for supporting this endeavor. Regarding evaluation, we have shown that our methodology is general enough to handle a broad range of type systems, precise enough to be implemented in Haskell, and we have proved that it produces correct type systems with respect to the gradual typing criteria.

Acknowledgments

We thank Sam Tobin-Hochstadt, Michael M. Vitousek, Andre Kuhlenschmidt, Andrew Kent, Ambrose Bonnaire-Sergeant, Ronald Garcia and the anonymous reviewers for their feedback on this work. This work is partially funded by NSF grant 1518844.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] A. Ahmed, R. B. Findler, J. Matthews, and P. Wadler. Blame for all. In *Workshop on Script to Program Evolution (STOP)*, July 2009.

[3] A. Ahmed, R. B. Findler, J. G. Siek, and P. Wadler. Blame for All. In *Symposium on Principles of Programming Languages*, January 2011.

[4] E. Allende, O. Callaú, J. Fabry, É. Tanter, and M. Denker. Gradual typing for Smalltalk. *Science of Computer Programming*, Aug. 2013. Available online.

[5] G. Bierman, E. Meijer, and M. Torgersen. Adding dynamic types to C#. In *European Conference on Object-Oriented Programming*, ECOOP'10. Springer-Verlag, 2010.

[6] G. Bierman, M. Abadi, and M. Torgersen. Understanding TypeScript. In R. Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, volume 8586 of *Lecture Notes in Computer Science*, pages 257–281. Springer Berlin Heidelberg, 2014.

[7] S. K. Debray and D. S. Warren. Automatic mode inference for logic programs. *J. Log. Program.*, 5(3):207–229, Sept. 1988. ISSN 0743-1066. .

[8] T. Disney and C. Flanagan. Gradual information flow typing. In *Workshop on Script to Program Evolution*, 2011.

[9] R. Garcia. Calculating threesomes, with blame. In *ICFP '13: Proceedings of the International Conference on Functional Programming*, 2013.

[10] R. Garcia and M. Cimini. Principal type schemes for gradual programs. In *Symposium on Principles of Programming Languages*, POPL, pages 303–315, 2015. .

[11] R. Garcia, A. M. Clark, and É. Tanter. Abstracting gradual typing. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St Petersburg, FL, USA, January 20-22*. ACM Press, 2016.

[12] A. Hejlsberg. Introducing TypeScript. Microsoft Channel 9 Blog, 2012.

[13] D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. In *Trends in Functional Prog. (TFP)*, page XXVIII, April 2007.

[14] D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. *Higher Order Symbol. Comput.*, 23(2):167–189, June 2010. ISSN 1388-3690.

[15] L. Ina and A. Igarashi. Gradual typing for generics. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, 2011.

[16] D. Jacobs. A pragmatic view of types for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, Logic programming, pages 217–227. MIT Press, 1992.

[17] E. Meijer and P. Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, 2004.

[18] D. Miller and G. Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, New York, NY, USA, 1st edition, 2012. ISBN 052187940X, 9780521879408.

[19] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[20] U. S. Reddy. A typed foundation for directional logic programming. In E. Lamma and P. Mello, editors, *Third International Workshop on Extensions of logic programming*, pages 150–167, Bologna, Italy, 1993. Springer-Verlag LNAI 660.

[21] J. Siek, M. Vitousek, M. Cimini, and J. Boyland. Refined Criteria for Gradual Typing. In *Proceedings of the Inaugural Summit oN Advances in Programming Languages (SNAPL 2015)*, May 2015. To appear.

[22] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, pages 81–92, September 2006.

[23] J. G. Siek and W. Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming*, volume 4609 of *LCNS*, pages 2–27, August 2007.

[24] J. G. Siek and M. Vachharajani. Gradual typing and unification-based inference. In *DLS*, 2008.

[25] J. G. Siek and P. Wadler. Threesomes, with and without blame. In *Symposium on Principles of Programming Languages*, POPL, pages 365–376, January 2010.

[26] J. G. Siek, R. Garcia, and W. Taha. Exploring the design space of higher-order casts. In *European Symposium on Programming*, ESOP, pages 17–31, March 2009.

[27] J. G. Siek, M. Vitousek, M. Cimini, S. Tobin-Hochstadt, and R. Garcia. Monotonic references for efficient gradual typing. In *European Symposium on Programming*, ESOP, 2015. To appear.

[28] J. G. Siek, M. M. Vitousek, M. Cimini, and J. T. Boyland. Refined criteria for gradual typing. In *SNAPL: Summit on Advances in Programming Languages*, LIPIcs: Leibniz International Proceedings in Informatics, May 2015.

[29] R. F. Stark. The declarative semantics of the prolog selection rule. In *SYMPOSIUM ON LOGIC IN COMPUTER SCIENCE*, pages 252–261. IEEE Computer Society Press, 1994.

[30] A. Takikawa, T. S. Strickland, C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Gradual typing for first-class classes. In *Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 793–810, 2012.

[31] A. Takikawa, D. Feltey, E. Dean, M. Flatt, R. B. Findler, S. Tobin-Hochstadt, and M. Felleisen. Towards practical gradual typing. In *European Conference on Object-Oriented Programming*, LIPICS. Dagstuhl Publishing, 2015.

[32] T. D. Team. *Dart Programming Language Specification*. Google, 1.2 edition, March 2014.

[33] P. Thiemann. Session types with gradual typing. In *International Symposium on Trustworthy Global Computing*, 2014.

[34] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *Symposium on Principles of Programming Languages*, January 2008.

[35] S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *International Conference on Functional Programming*, ICFP, pages 117–128. ACM, 2010.

[36] M. M. Vitousek, J. G. Siek, A. Kent, and J. Baker. Design and evaluation of gradual typing for Python. In *Dynamic Languages Symposium*, 2014.

[37] P. Wadler and R. B. Findler. Well-typed programs can't be blamed. In *European Symposium on Programming*, ESOP, pages 1–16, March 2009.

[38] R. Wolff, R. Garcia, E. Tanter, and J. Aldrich. Gradual typestate. In *European Conference on Object-Oriented Programming*, ECOOP'11. Springer-Verlag, 2011.