The Gradualizer: Gradual Typing for Free

Matteo Cimini Jeremy G. Siek
Indiana University
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 the benefits for a coherent and full-span migration between the two disciplines. However, the literature lacks a general methodology for turning statically-typed languages into gradually typed ones. Our main contribution is *the Gradualizer*: a general procedure that takes as input a type system and automatically produces the gradually typed version and the compilation procedure to the cast calculus. Our procedure handles a large class of type systems and we show the applicability of the Gradualizer by recovering a wealth of gradually typed systems and generating novel ones. We prove that the Gradualizer always generates systems that are correct with respect to three formal criteria. We also report on an implementation of the Gradualizer in Haskell that takes type systems expressed in λ-prolog and outputs their gradualized type system in λ-prolog.

1. Introduction

Many languages such as C#, Dart, Pyret, Racket, TypeScript and VB, to name a few, integrate static and dynamic typing. Siek and Taha [9] offered gradual typing as an approach to this integration that provides nice benefits for a coherent and full-span migration between the two disciplines.

However, designing a gradually typed calculus affording these benefits has proven to be suprisingly tricky. Language designers might have two main questions: *how do I lift my typed calculus to gradual typing?* and *how do I know that I have designed a good gradually typed calculus?* To help language designers, we are embarking on an ongoing research that aims at providing formal correctness criteria, general methodologies and automated tools for supporting the shift to gradual typing.

This paper is part of this effort and addresses the very first and essential steps on the road to a gradually typed calculus: how to derive a gradually typed system and how to derive the compilation to the cast calculus. These together forms the static aspects of gradual typing.

Unfortunately, the literature lacks of a general methodology for deriving a gradually typed calculus from a static one, and the only resources available are the examples of typed calculi that are gradualized by experts ([2, 5, 9, 10, 13, 14], for instance). These papers do put forward a few guidelines, however these guidelines are incomplete and do not provide a disciplined and general methodology. Mistakes often occur in the design of gradually typed languages [8]. The practical drawback for programmers is that these language designs typically provide less effective support for moving along the static-dynamic spectrum. We shall expand on this matter in Section 2.

How do we formulate a gradually typed calculus from a statically-typed one? Consider for instance the simply typed λ-calculus (STLC) and its gradually typed version (GTLC). GTLC achieves the integration of the dynamic type $\star$ by translating the typing rule for function application in the following way.

$$\frac{\Gamma \vdash\ e_1 : T_1 \to T_2 \quad \Gamma \vdash\ e_2 : T_1}{\Gamma \vdash\ (e_1\, e_2) : T_2} \quad \overset{\texttt{toGr}}{\Longrightarrow} \quad \frac{\Gamma \vdash_G\ e_1 : \star \quad \Gamma \vdash_G\ e_2 : T_1}{\Gamma \vdash_G\ (e_1\, e_2) : \star} \qquad \frac{\Gamma \vdash_G\ e_1 : T_1 \to T_2 \quad \Gamma \vdash_G\ e_2 : T_3 \quad T_1 \sim T_3}{\Gamma \vdash_G\ (e_1\, e_2) : T_2}$$

This transformation reveals a few useful patterns. For instance, the upper rule on the right sets the very first guide line of gradual typing, i.e., we must *1) create a rule in which the function type is replaced with* $\star$. The bottom rule on the right shows that *2) uses of type equality must be replaced with consistency (*$\sim$*).*

However, the STLC is a particularly simple type system that involves only a few of the possible scenarios that can occur in typing rules. The guidelines that we find in GTLC, and in the literature to date, leave open questions on how to deal with arbitrary type systems. In this paper we answer the following questions.

Question 1: When should complex types be replaced by $\star$*?* Consider the typing rule for objects in the object calculus of Abadi and Cardelli [1].

$$\frac{\mathit{for}\ 1 \le i \le k : \quad \Gamma, \mathit{self} : (\mathit{obj}\ [l_i, T_i]) \vdash\ e_i : T_i}{\Gamma \vdash\ (\mathit{create}\ [l_i, T_i, e_i])\ (\mathit{obj}\ [l_i, T_i])}$$

Does guideline 1) apply also to $\mathit{self} : (\mathit{obj}\ [l_i, T_i])$? Should we duplicate this rule for accommodating the case of $\star$? It turns out that the gradually typed object calculus of [10] does not. Why does this occurence of a complex type get a different treatment compared to the function type in the application rule? Also, why do not replace complex types with $\star$ in the conclusion of a typing rule?

Question 2: To which variables should we apply consistency? Consider the case expression of sum types, below on the left.

$$\frac{\Gamma \vdash\ e_1 : (T_1 + T_2) \quad \Gamma, x : T_1 \vdash\ e_2 : T \quad \Gamma, x : T_2 \vdash\ e_3 : T}{\Gamma \vdash\ (\mathit{case}\ e_1\ e_2\ e_3) : T} \quad \overset{\texttt{toGr}}{\Longrightarrow} \quad \frac{\Gamma \vdash\ e_1 : (T_1 + T_2) \quad \Gamma, x : T_1 \vdash\ e_2 : T \quad \Gamma, x : T_2 \vdash\ e_3 : T' \quad T \sim T'}{\Gamma \vdash\ (\mathit{case}\ e_1\ e_2\ e_3) : T^{?}}$$

Type equality is used multiple times in this rule. Should we have two versions of the variable T_1 related by consistency? Should we have two versions of T_2 and three versions of T? It turns out that only the occurences of T should be treated with consistency and only those occurrences that are in the premises. Why this special treatment for T and not for T_1 and T_2?

Question 3: Which variables are to be replaced and with what? Consider the case statement above and imagine that we had separated the two T's into two new variables T' and T''. Following the treatment in [4] for the similar construct `if–then–else`, we need to assign the join of them to the T in the conclusion, therefore $T^? = (T \sqcap T')$ in the rule above. Why among three occurrences only this latter is not involved in the calculation of the join and is replaced by the join type? Also, is the join type always the correct substitution?

Question 4: When must variables only range over static types? Consider the typing rule for abstraction for the implicitly typed lambda-calculus (ITLC).

$$\frac{\Gamma, x : T_1 \vdash\ e : T_2}{\Gamma \vdash\ \lambda x.e : T_1 \to T_2}$$

Were we to apply the same treatment as for STLC, its gradually typed system would have exactly this same rule (but using $\vdash_G$). This is an incorrect treatment, as argued in [11], because T_1 comes out of nowhere and can be instantiated with $\star$ which allows some programs to type check when they should not. Following [4], the gradual type system must ensure that T_1 ranges only over static types. But why do we place this constraint on T_1 but not T_2?

In this paper, we develop a unified methodology to answer these questions and, in general, for *gradualizing* type systems. The discipline is based on the *input/output modes* of the various relations used in the rules and on the *input/output* capabilities of the type constructors of the language. We walk the reader through all the steps from a static type system to a gradual type system, explaining the steps of our methodology with examples. We also show that deviating from the prescribed steps can jeopardize the correctness of the resulting gradual type system.

Typically, gradually typed calculi are compiled into a cast calculus for execution. As we shall discuss in Section 2, this compilation also comes with its own open questions for language designers. Our next contribution is the application of our approach to generating the compilation function. Along similar lines as for type systems, we develop a methodology for deriving the procedure of the compilation.

We make our methodology completely precise and automatic in the form of *the Gradualizer*: a general procedure that takes a type system as input, represented as a logic program of 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 manipulating logic programs. Using λ-prolog is a convenient vehicle for applying our methodology to a very general class of type systems. The Gradualizer is depicted in Figure 1.

We offer an implementation of the Gradualizer. To be clear on what our tool does and how to use it, consider Figure 2. That is, from the type checker of the original system (STLC in Figure 2), we generate the type checker for its gradually typed version (`typeOfGr`) and for its cast calculus (`typeOfCC`). Also, we generate the logic program for the compilation to the cast calculus (`compToCC`). Because the results are logic programs, they can be interrogated with queries for typeability of programs or compilation. We have applied the Gradualizer and our tool to a wealth of examples. In fact, we have gradualized many of the type systems of Pierce [7], namely: STLC, unit type, pairs, tuples, records, let binding, general recursion (fix), sum types, exceptions, references, lists and STLC with subtyping.

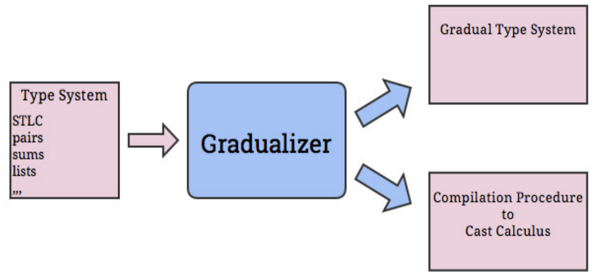

Figure 1. The Gradualizer

Of course, the typed calculi generated by the Gradualizer are useful only as long as they are *correct*, and what does it mean for these type systems to be correct? In this regard, this is possibly the best moment for supporting tools for gradual typing, as [8] recently has drawn a crisp analysis on what formal properties gradual systems must afford and so we have precise standards for our Gradualizer to meet. We indeed prove that the Gradualizer always generates typed calculi that are correct with respect to three criteria.

In summary, we make the following contributions.

1. A novel and correct methodology for creating the static semantics of gradually-typed languages. For the first time, we give an explicit walkthrough on the process of gradualizing 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 many of the type systems of Pierce [7]. Some of the resulting gradual type systems are novel (Section 4).
3. The Gradualizer: an automatic procedure that takes a type system as input and produces its gradually typed counterpart (Section 6). We also provide an implementation of the Gradualizer in Haskell. Our tool can produce typecheckers and compilers in λ-prolog.
4. A proof that the type systems generated by the Gradualizer always satisfy three formal correctness criteria for gradual typing (Section 7).

2. Overview on Gradual typing

The road to gradual typing encompasses quite a few systems. It is useful to depict below the typed calculi involved in this landscape and provide an overview on how they are related to each other.

Type System $\mathbb{T}$	Gradual T. S. $\mathbb{T}^G$	Cast Calculus $\mathbb{T}^{CC}$
Exp	Exp	Exp + cast
Types	Types + $\star$	Types + $\star$
$\vdash$	$\vdash_G$	$\vdash_{CC}$ = $\vdash$ + cast

Gradually typed system: $\mathbb{T} \overset{\texttt{toGr}}{\Longrightarrow} \mathbb{T}^G$ When integrating dynamic typing[1] to a type system $\mathbb{T}$, the language of expressions of $\mathbb{T}$ is not extended, i.e. the set of operators of the language remains the same. The addition, however, is made on the set of types that now is augmented with the dynamic type $\star$. Of course, this addition must be accompained with a proper treatment of $\star$. For instance, in GTLC we would like that functions that accept arguments of the dynamic type indeed let the parameter passing happen for integers,

[1] Whether it is a statically typed language to be integrated with dynamic features or the other way around, the language designer must have a static type system in mind and gradual typing acts on that.

```
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.

typeOfGr (abs T1 R) ( arrow T1 T2) :- (pi x\ ( typeOfGr x T1 => typeOfGr (R x) T2)).
typeOfGr (app E1 E2) T2 :- typeOfGr E1 PMV1, matchGrarrow PMV1 T1 T2,
                           typeOfGr E2 TT2, consistency T1 TT2.

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

compToCC (abs T1 R) (abs T1 R') (arrow T1 T2) :-
                                  (pi x\ (compToCC x x T1 => compToCC (R x) (R' x) T2 )).
compToCC (app E1 E2) ( app (cast PMV3 (arrow T1 T2) LabelLL5 E1') (cast TT4 T1 LabelLL6 E2')) T2
                        :- compToCC E1 E1' PMV3, matchGrarrow PMV3 T1 T2,
                           compToCC E2 E2' TT4, consistency T1 TT4.
```

Figure 2. Input/output of the implementation of the Gradualizer.

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

$$\frac{\Gamma \vdash_G e_1 : X \qquad =_\to X\, T_1\, T_2 \qquad \Gamma \vdash_G e_2 : T_3 \qquad T_1 \sim T_3}{\Gamma \vdash_G (e_1\, e_2) : T_2}$$

$\texttt{Int} \sim \star \quad \texttt{Bool} \sim \star$

$$\frac{T_1 \sim T_3 \qquad T_2 \sim T_4}{T_1 \to T_2 \sim T_3 \to T_4} \qquad =_\to (T_1 \to T_2)\, T_1\, T_2$$

$\sim$ is reflexive and symmetric. $\qquad =_\to \star\ \star\ \star$

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

i.e. that the program $(\lambda x : \star.x)\ 4$ would be typeable. Gradual typing achieves this scenario thanks to the help of the consistency relation $\sim$. As an example, Figure 3 shows the translation from STLC into GTLC (we will talk shortly about its formulation style) and shows the definition of consistency for the types of STLC.

The type system $\vdash_G$ employs consistency at proper points in the type system. The usage of consistency applies more liberal checks and are responsible for the correct interplay of $\star$ with the other types in the different contexts of $\mathbb{T}^G$. The result is a type system that still rejects programs when inconsistencies are found and optimistically let expressions type-check on the ground of consistency.

Notice that the formulation of Figure 3 differs from the original presentation of [9]. Indeed, it follows a style first appeared in [4], though in a different form, and avoids the duplication of rules thanks to pattern-matching premises of the form $=_\to X\ T_1\ T_2$. The two bottom-right lines of Figure 3 define when these premises are satisfied. Basically, these premises pattern-matches the type X and if it is a function type its domain and codomain are returned. When X is $\star$, the second clause realizes a well-known tagline for gradual typing: *the arrival of $\star$ in lieu of a function type must be treated as $\star \to \star$*. If X is any other type the premise simply fails to hold, and the overall rule will not be applicable.

Compilation to the cast calculus: $\mathbb{T}^G \stackrel{\texttt{toCI}}{\Longrightarrow} \mathbb{T}^{CC}$ How can we execute programs in $\mathbb{T}^G$? An option would be to use a fully dynamic version of the operational semantics of the original calculus. However, unnecessary run-time checks would be ubiquitous, affecting the performance of the language overall. We would like to take advantage of the information of those types that at least are known at compile time, as they would allow for an efficient execution.

What typically happens in gradual typing is that programs of $\mathbb{T}^G$ are compiled into a cast calculus $\mathbb{T}^{CC}$, yet another typed calculus with its own type system $\vdash_{CC}$. The set of types of $\mathbb{T}^{CC}$ includes $\star$ and the expressions language is extended with an explicit operator for performing casts. There has been a few proposals for cast operators in the context of gradual typing and the adoption of one over another affects the dynamic semantics and its blame tracking aspects. For the scope of the static semantics, the choice of the cast operator is irrelevant and we adopt twosome casts of [12], though with the more convenient notation $e : T_1 \Rightarrow^l T_2$, where l is a label. The work carried out in this paper can be easily adapted to other shapes of casts.

The compilation to $\mathbb{T}^{CC}$ is denoted here by $\vdash_{CC} \hookrightarrow$ and used as $\vdash_{CC}\ \ e \hookrightarrow e' : T$ meaning that the program e is compiled into e' and it has type T. For brevity, we sometimes simply write $\hookrightarrow e' : T$ when the rest is clear, sometimes also omitting T when irrelevant. This compilation, also known as *cast insertion*, inserts appropriate run-time cast checks at the points where the gradual type system could not clearly resolve types. Out of consistency, some expressions has passed the checks at compile time but these checks will be performed at run-time by the cast calculus. The compilation is derived from $\vdash_G$, by way of example the following cast insertion is for STLC.

$$\frac{\Gamma \vdash_G e_1 : X \qquad =_\to X\, T_1\, T_2 \qquad \Gamma \vdash_G e_2 : T_3 \qquad T_1 \sim T_3}{\Gamma \vdash_G (e_1\, e_2) : T_2}$$

$$\stackrel{\texttt{toCI}}{\Longrightarrow}$$

$$\frac{\Gamma \vdash_{CC} e_1 \hookrightarrow e_1' : X \qquad =_\to X\, T_1\, T_2 \qquad \Gamma \vdash_{CC} e_2 \hookrightarrow e_2' : T_3 \qquad T_1 \sim T_3}{\hookrightarrow ((e_1' : X \Rightarrow^{l_1} T_1 \to T_2)\ (e_2' : T_3 \Rightarrow^{l_2} T_1)) : T_2}$$

Notice that the set of premises of the original rule and the translated one always coincide modulo the use of $\vdash_{CC} \hookrightarrow$ in lieu of $\vdash_G$. Therefore, for brevity we shall display only the conclusion for the rule translated by $\stackrel{\texttt{toCI}}{\Longrightarrow}$. What the compilation concretely does, and that shows another guideline from the literature, is that *every uses of up-to-$\star$ features must translate into run-time cast checks in the compiled program*.

Open questions on $\overset{\texttt{toCI}}{\Longrightarrow}$ Unfortunately, also this guideline is quite vague. Consider the cast insertion for the case constructor of sum types where the cast targets the join type (below, e_1^* contains a cast as well but it is not relevant to our point and not shown).

$$\frac{\begin{array}{cc}\Gamma \vdash\ e_1 : (T_1 + T_2) & \Gamma, x : T_1 \vdash\ e_2 : T \\ T \sim T' & \Gamma, x : T_2 \vdash\ e_3 : T'\end{array}}{\Gamma \vdash\ (\mathit{case}\ e_1\ e_2\ e_3) : T^j}$$
$$\overset{\texttt{toCI}}{\Longrightarrow}$$
$$\hookrightarrow (\mathit{case}\ e_1^*\ (e_2' : T \Rightarrow^{l_1} T^j)\ (e_3' : T' \Rightarrow^{l_2} T^j)) : T^j$$

<u>Question 5</u>*: Is there a general rule for understanding where to cast?* Sometimes the correct cast is to the join type while some other is to a type variable that is related by consistency. In this latter case, which one to pick when many variables are related by consistency? It turns out that this matter is closely related to Question 3. We will discuss this in Section 3.

The type system of the Cast Calculus: $\mathbb{T} \overset{\texttt{toCC}}{\Longrightarrow} \mathbb{T}^{CC}$ Even though the compilation $\vdash_{CC} \hookrightarrow$ is derived from $\vdash_G$, the type system of the cast calculus does not look at all like this latter. Indeed, the cast calculus is meant to simply be the original calculus augmented with a cast operator and the dynamic type. The type system $\vdash_{CC}$ is a straightforward extension of the type system $\vdash$ with a typing rule for the cast operator.

$$\vdash \quad \overset{\texttt{toCC}}{\Longrightarrow} \quad (\vdash \cup \frac{\Gamma \vdash\ e : T_1}{\Gamma \vdash\ (e : T_1 \Rightarrow^l T_2) : T_2})$$

<u>Our main goal:</u> This paper is set to provide a general methodology for the translations $\overset{\texttt{toGr}}{\Longrightarrow}$ and $\overset{\texttt{toCI}}{\Longrightarrow}$, and to provide a set of procedures for automatically perform them on a large class of type systems: the Gradualizer.

Correctness criteria for gradual typing What are the properties that a gradually typed calculus must have w.r.t. its original calculus? This question is at the core of the foundational matter *what is gradual typing?* The recent work [8] explicitly addresses this matter and puts forward a set of correctness criteria for gradual typing. We here review only the criteria that are relevant to this paper, therefore, those related to the static aspects.

One of the first properties to check is that programs that are typeable in the original calculus be typeable also in the gradual calculus. Furthermore, it would be strongly ill-behaved for the gradual calculus to start typing programs that are untypeable in the original calculus. The first criterion is therefore that their typeability must coincide over *static* programs, i.e., they do not contain any $\star$.

$$\textit{for all static } e \textit{ and } T, \vdash\ e : T \textit{ if and only if} \vdash_G\ e : T. \qquad (1)$$

However, this criterion alone does not tell the whole story. Consider the expressions in Figure 4 where nodes at lower levels of the lattice are less precise, i.e., roughly, they contain more $\star$.

One would like that the programs in blue would *all* be typeable. This mirrors the expecation that, when removing type annotations, a well-typed program will continue to be well-typed (with no need to insert explicit casts). This also implies that programs in red would *all* be untypeable, a pleasant property for programmers as they would spot a wrong path right at the first mistaken type annotation. To cover these aspects, [9] puts forward the following *static gradual guarantee* that requires $\vdash_G$ to be monotonic over the less-precision relation $\sqsubseteq$.

$$\begin{array}{c}\textit{for all } e, e', T, \textit{ if} \vdash_G\ e : T \textit{ and } e' \sqsubseteq e \textit{ then} \\ \textit{it exists } T' \textit{ such that} \vdash_G\ e' : T' \textit{ and } T' \sqsubseteq T.\end{array} \qquad (2)$$

Criteria (1-2) are fundamental for gradual typing at a very foundational level. It explains for instance why both the programs $(\lambda x : Int)\ 4$ and $(\lambda x : \star. x)\ 4$ *must* be typeable in GTLC, being the former typable in STLC and the latter a less-precise version of it.

The compilation to the cast calculus must as well conform to some criteria. In particular, the cast insertion must exist for typeable programs and must be type preserving.

$$\textit{for all } e, e', T, \textit{ if} \vdash_G\ e : T \textit{ then} \vdash_{CC}\ e \hookrightarrow e' : T \textit{ and} \vdash_{CC}\ e' : T \qquad (3)$$

In this paper, Criteria (1-3) will set the bar for our Gradualizer to meet. We have proved indeed that the Gradualizer always generates gradually typed calculi that satisfy (1-3). It is important to notice, however, that [9] points out one more important property: the compilation must be monotonic w.r.t. the less-precision relation. We will argue in Section 7 that in order to address this property we shall need a deeper analysis of some aspect of the Gradualizer. We leave such analysis for future work.

3. Methodology for Gradualizing Languages

In this section, we provide a unified methodology to the process of gradualizing. The methodology is based on the *input/output modes* of the various relations involved in the rules. For instance, the assertion $\vdash\ e\ T$ clearly makes this distinction: the program e is an input while the type T is an output. Pattern-matching premises as $(=_\rightarrow\ X\ T_1\ T_2)$ also make this distinction: X is the input to be pattern-matched while T_1 and T_2 are the outputs. In general, we define a function $\texttt{mode}(rel, k) = m$, where $m \in \{i, o\}$, that intuitively say whether the k-th argument of the relation rel is an input or an output. The methodology is also based on the *input/output capabilities* of the arguments of type constructors, for which we define the function $\texttt{capab}(f, k) = m$. For instance, in the type $T_1 \rightarrow T_2$, T_1 is an input and T_2 is an output. In our terminology, T_1 is a *type-input* and T_2 is a *type–output*. As an abbreviation, we shall write $\texttt{mode}(=_\rightarrow) = (i, o, o)$ and use this notation accordingly for defining $\texttt{mode}$ on other predicates and also for defining $\texttt{capab}$. Below we have modes and capabilities for a few relations and type constructors encountered so far.

$$\begin{array}{llll}\texttt{mode}(\vdash) & = (i, o) & \texttt{mode}(=_\rightarrow) = (i, o, o) \\ \texttt{mode}(\vdash_{CC} \hookrightarrow) & = (i, o, o) & \texttt{mode}(<:) = (i, o) \\ \texttt{capab}(\rightarrow) & = (i, o) & \texttt{capab}(+) = (o, o) \\ \\ \texttt{mode}^{-1}(rel, k) & = m^{-1} & \texttt{capab}^{-1}(f, k) = m^{-1} \\ \\ & i^{-1} = o & o^{-1} = i\end{array}$$

Notice that the functions $\texttt{mode}$ and $\texttt{capab}$ account for the input-output flavor of their arguments when ordinary calls are made to it, that is, when they appear in ordinary premises of a rule. However, terms might appear on the other side of an implicit implication, namely, in the conclusion and in the type environment. In those cases, the input-output results are swapped, hence the definition of $\texttt{mode}^{-1}$. That an implicit implication is crossed by $\vdash$ when stepping into the type environment will be explained in Section 5. To help our presentation, in the rest of the paper we will give a color to some variables. The convention that we will follow is that blue variables will be outputs , red variables will be input and cyan variables will be type-input outputs . Sometimes we will avoid coloring *all* the variables only to do it for the relevant ones. By way of example, here below are two rules.

$$\begin{tikzcd}[column sep=tiny]
& (\lambda y{:}\star.\,y)\ ((\lambda x{:}\star.\,x)\ 42) \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\
(\lambda y{:}\texttt{Int}.\,y)\ ((\lambda x{:}\star.\,x)\ 42) \arrow[d, dash] & (\lambda y{:}\star.\,y)\ ((\lambda x{:}\texttt{Int}.\,x)\ 42) \arrow[dl, dash] & (\lambda y{:}\texttt{Bool}.\,y)\ ((\lambda x{:}\star.\,x)\ 42) \arrow[d, dash] \\
(\lambda y{:}\texttt{Int}.\,y)\ ((\lambda x{:}\texttt{Int}.\,x)\ 42) & & (\lambda y{:}\texttt{Bool}.\,y)\ ((\lambda x{:}\texttt{Bool}.\,x)\ 42)
\end{tikzcd}$$

Figure 4. A lattice of differently-annotated versions of a gradually-typed program.

A zoom into $\mathbb{T} \overset{\texttt{toGr}}{\Longrightarrow} \mathbb{T}^G$:

$$\mathbb{T} \overset{\texttt{toPM}}{\Longrightarrow} \mathbb{T}^p \overset{\texttt{toCnst}}{\Longrightarrow} \mathbb{T}^c \overset{\texttt{toRes}}{\Longrightarrow} \mathbb{T}^r \overset{\texttt{toSt}}{\Longrightarrow} \mathbb{T}^G$$

And into $\mathbb{T}^G \overset{\texttt{toCI}}{\Longrightarrow} \mathbb{T}^{CC}$:

$$\mathbb{T}^G \overset{\texttt{toRes}}{\Longrightarrow} \mathbb{T}^{Gr} \overset{\texttt{toCast}}{\Longrightarrow} \mathbb{T}^{CC}$$

Figure 5. Fine-grained view of $\overset{\texttt{toGr}}{\Longrightarrow}$ and $\overset{\texttt{toCI}}{\Longrightarrow}$.

$$\frac{\begin{array}{c}\Gamma \vdash\ e_1\ (T_1\ + T_2\,) \\ \Gamma, x : T_1\ \vdash\ e_2\ T \\ \Gamma, x : T_2\ \vdash\ e_3\ T\end{array}}{\Gamma \vdash\ (case\ e_1\ e_2\ e_3) : T} \qquad \frac{\begin{array}{c}\Gamma \vdash\ e_1 : T_1\ \to T_2 \\ \Gamma \vdash\ e_2 : T_1\end{array}}{\Gamma \vdash\ (e_1\ e_2) : T_2}$$

Type-input outputs Type-inputs will guide our methodology towards understanding where inputs should receive their values from. Notice that type-inputs appear when the language designer expands the structure of an output. Revealing such structure, now we need to know extra information if we want to make use of its internal types. In our methodology, type-inputs are such that all the other occurrences of the same variable points at it as final type. For correctness, a type can appear in type-input position at most once (see Example in Section 3.3).

Gradual type system The series of questions that we have raised in the introduction provide a nice starting point for a methodology of gradual typing. For the gradual type system, in fact, our methodology is accomplished in four steps, each of which answers one of those questions. Figure 5 depicts the entirety of this process. As a quick reference, here below we accompany each step with an extended guideline that refines what we encounter in the literature. The following sections will expand on these steps with details and examples.

- Pattern-matching of outputs ($\overset{\texttt{toPM}}{\Longrightarrow}$, Section 3.1): Prepares the type system to accept types that might not appear in their canonical type, rather, injected into the dynamic type $\star$. This acts only on types in output position and makes use of pattern-matching.
- Up-to consistency for type-output outputs ($\overset{\texttt{toCnst}}{\Longrightarrow}$, Section 3.2): Makes explicit the usage of type equality checks only for certain types. In particular, only output types that are also type-outputs are considered up-to consistency.
- Resolution of inputs ($\overset{\texttt{toRes}}{\Longrightarrow}$, Section 3.3): Inputs that do not have an occurrence as type-input are replaced by the join type of their outputs that have been considered up-to consistency.
- Ensuring staticity for free inputs ($\overset{\texttt{toSt}}{\Longrightarrow}$, Section 3.4): Enforces input types that are free, i.e., they do not receive a value from any output, to be static in the gradual type system.

Compilation to the Cast Calculus One of the hardest part for a general methodology for gradual typing is answering Question 3: *Which variables are to be replaced and with what?* and Question 5: *Is there a general rule for understanding where to cast?* Fortunately, these questions are closely related and our answers for the former provides the necessary insights for applying a correct compilation to the cast calculus. The compilation to the cast calculus is achieved in two step.

- Encoding with Resolution of inputs ($\overset{\texttt{toRes}}{\Longrightarrow}$, Section 3.5): As the encoding term in the conclusion is in input position, output variables now are inputs and must be resolved.
- Cast to the final type. ($\overset{\texttt{toCast}}{\Longrightarrow}$, Section 3.5): The encoding variables are first cast to the expansion of their type (if their type is a pattern-matching variable), and then cast to a same type that uses the final type for those outputs that are not type-inputs.

As the first the step is derived by $\overset{\texttt{toRes}}{\Longrightarrow}$ and easy to explain, we will discuss the full compilation in one section.

3.1 Step 1: Pattern-matching of outputs

> $\mathbb{T} \overset{\texttt{toPM}}{\Longrightarrow} \mathbb{T}^p$: Prepares the type system to accept types that might not appear in their canonical type, rather, injected into the dynamic type $\star$. This acts only on types in output position and makes use of pattern-matching.

As an example, we shall see the application of this guideline to the typing rule for application of STLC.

$$\frac{\begin{array}{c}\Gamma \vdash\ e_1 : T_1 \to T_2 \\ \Gamma \vdash\ e_2 : T_1\end{array}}{\Gamma \vdash\ (e_1\ e_2) : T_2} \overset{\texttt{toPM}}{\Longrightarrow} \frac{\begin{array}{c}\Gamma \vdash_G\ e_1 : X \quad\ =_{\to} X\ T_1\ T_2 \\ \Gamma \vdash_G\ e_2 : T_1\end{array}}{\Gamma \vdash_G\ (e_1\ e_2) : T_2}$$

Since the type in blue is complex and it is in output position for $\vdash$, we apply pattern-matching to it.

The reader should notice that type constants such as `Int` and `Bool` are type constructors of arity 0 and they will be pattern-matched if found in output positions. Consider the following example.

$$\frac{\begin{array}{c}\Gamma \vdash\ e_1 : \texttt{Int} \\ \Gamma \vdash\ e_2 : \texttt{Int}\end{array}}{\Gamma \vdash\ e_1 + e_2 : \texttt{Int}} \overset{\texttt{toPM}}{\Longrightarrow} \frac{\begin{array}{c}\Gamma \vdash\ e_1 : X \quad\ =_{\texttt{Int}}\ X \\ \Gamma \vdash\ e_2 : X \quad\ =_{\texttt{Int}}\ X\end{array}}{\Gamma \vdash\ e_1 + e_2 : \texttt{Int}}$$

$$=_{\texttt{Int}}\ \texttt{Int} \qquad =_{\texttt{Int}}\ \star$$

Notice that this matches exactly the treatment for `Int` in the analogous rule for $+$ in [4], modulo our usage of pattern-matching.

The types that are in the conclusion and being proved are sometimes complex types. An example is the red occurrence of `Int` in the rule above. However, because of the input-output swap they are in input positions and they are never pattern-matched. For analogous reasons, complex types that appear in the type environment are not pattern-matched.

Complex types might be nested at will and pattern-matching needs to pattern-match also their subterms. Consider for instance a language with exceptions and the following typing rule for the try constructor.

$$\frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : \texttt{ExcType} \rightarrow T}{\Gamma \vdash \mathit{try}\ e_1\ \mathit{with}\ e_2 : T} \overset{\texttt{toPM}}{\Longrightarrow} \frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : X \quad =_{\rightarrow} X\ Y\ T \quad =_{\texttt{ExcType}} Y}{\Gamma \vdash \mathit{try}\ e_1\ \mathit{with}\ e_2 : T}$$

$$=_{\texttt{ExcType}} \texttt{ExcType} \qquad =_{\texttt{ExcType}} \star$$

Violating our discipline

Example 1: Not pattern-matching on output variables. Assume that we were not to act on the type $T_1 \rightarrow T_2$ and that GTLC would have the same typing rule as STLC for application. Then, the program $\lambda x : \star.(x\ 3)$ would not be typeable because $\star$ would fail to pattern-match with the complex pattern $T_1 \rightarrow T_2$ for the subterm $x\ 3$. However, the program $\lambda x : (\texttt{Int} \rightarrow \texttt{Int}).(x\ 3)$ is typeable and the former is simply a less-precise version of it. Therefore, Criteria (2) is violated.

Example 2: Pattern-matching on input variables. Consider a specialized abstraction λ^i that types only functions from integers to integers.

$$\frac{\Gamma, x : \texttt{Int} \vdash e : Int}{\Gamma \vdash \lambda^i\, x.e : \texttt{Int} \rightarrow \texttt{Int}} \overset{\texttt{toPM}}{\Longrightarrow} \frac{\Gamma, x : Y \vdash e : X \quad =_{\texttt{Int}} X \quad =_{\texttt{Int}} Y}{\Gamma \vdash \lambda^i\, x.e : \texttt{Int} \rightarrow \texttt{Int}}$$

As Y is free to be instantiated with $\star$ (consistent with $\texttt{Int}$), the program $let\ x = 4\ in\ (x\ x)$ is typeable by $\vdash_G$. However, this latter is a static program that is untypeable by $\vdash$. Criteria (1) is therefore violated.

Example 3: Not pattern-matching nested complex types. Consider the rule above for the try constructor for exception types. Assume that we were only pattern-matching the top-level $\rightarrow$, and have the premises $\Gamma \vdash\ e_2 : X$ and $=_{\rightarrow}\ X\ \texttt{ExcType}\ T$. Then, the program $try\ 3\ with\ \lambda x : \star.3$ would not be typeable, because X would be $\star \rightarrow \texttt{Int}$ and $=_{\rightarrow}\ (\star \rightarrow \texttt{Int})\ \texttt{ExcType Int}$ does not hold. However, the program $try\ 3\ with\ \lambda x : \texttt{ExcType}.3$ is typeable and the former is a less-precise version of it. Therefore, Criteria (2) is violated.

3.2 Step 2: Up-to consistency for type-output outputs

> $\mathbb{T}^p \overset{\texttt{toCnst}}{\Longrightarrow} \mathbb{T}^c$: Makes explicit the usage of type equality checks only for certain types. In particular, only output types that are also type-outputs are considered up-to consistency.

In order to set variables up-to consistency, we involve them in a k-ary join operator $T_1 \sqcap \ldots \sqcap T_n$. This also allows the usage of unary join that acts as the identity function and that we generally omit writing. The join operator considers $\star$ as its bottom, we have for instance $(\texttt{Int} \rightarrow \star) \sqcap (\star \rightarrow \texttt{Int}) = \texttt{Int} \rightarrow \texttt{Int}$. Notice also that the existence of a join is equivalent to consistency, i.e., $T_1 \sim T_2$ if and only if $\exists T^j.\, T_1 \sqcap T_2 = T^j$.

As an example of application of the guideline of this section, consider the case constructor.

$$\frac{\Gamma \vdash e_1\ (T_1 + T_2) \quad \Gamma, x : T_1 \vdash e_2\ T \quad \Gamma, x : T_2 \vdash e_3\ T}{\Gamma \vdash (case\ e_1\ e_2\ e_3) : T} \overset{\texttt{toCnst}}{\Longrightarrow} \frac{\Gamma \vdash e_1\ X \quad =_{+}\ X\ T_1'\ T_2' \quad \Gamma, x : T_1 \vdash e_2\ T' \quad \Gamma, x : T_2 \vdash e_3\ T'' \quad T' \sqcap T'' = T^j}{\Gamma \vdash (case\ e_1\ e_2\ e_3) : T}$$

The occurrences of T_1 and T_2 in the type environment are inputs, as it is the T in the conclusion. Therefore, these occurrences have not changed in the transformed rule. On the other hand, T appear twice as output, hence the treatment for it with the join calculation. To be precise, in the rule above also T_1' (and, separately, also T_2') is up-to consistency in an implicit unary join.

This step is fundamental to prepare the terrain for the resolution of inputs and a correct cast insertion. We therefore introduce another task for this step: *if there exists a type-input outputs, the join of the separated outputs must be consistent with this type-input output*. As we have said, this latter is unique. Consider for instance STLC with algorithmic subtyping.

$$\frac{\Gamma \vdash_G\ e_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash_G\ e_2 : T_3 \quad T_3 <: T_1}{\Gamma \vdash_G\ (e_1\ e_2) : T_2} \overset{\texttt{toCnst}}{\Longrightarrow} \frac{\Gamma \vdash\ e_1 : X \quad =_{\rightarrow}\ X\ T_1\ T_2' \quad \Gamma \vdash_G\ e_2 : T_3' \quad T_3 <: T_1' \quad T_1 \sim T_1'}{\Gamma \vdash_G\ (e_1\ e_2) : T_2}$$

The input-output mode for subtyping is $\texttt{mode}(<:) = (i, o)$. Also, with in mind $\texttt{capab}(\rightarrow)$, the upper occurrence of T_1 is a type-input. Because of this, those occurrences of T_1 and T_3 are left unchanged. All the other variables are then unary joins, in particular, T_1' is the join type of itself and it is checked for consistency with its type-input of reference T_1. Notice that this step leaves rules 'open' or 'unfinished'. The next step will fix this situation.

Violating our discipline

Example 1: Not up-to consistency for output variables. Assume that we were not to act on the variable T_1. This means that GTLC would have the same typing rule on the left-hand side of the translation. Then, the program $(\lambda x : \star.x)\ 4$ would not be typeable. However, the program $(\lambda x : \texttt{Int}.x)\ 4$ is typeable and is simply a less-precise version of the former. Criteria (2) is therefore violated.

Example 2: Up-to consistency for input variables. Consider the typing rule for abstraction of STLC. As T_1 appears twice in input (in the hypothetical premise and in the conclusion), assume that we were to consider it up-to consistency.

$$\frac{\Gamma, x : T_1' \Rightarrow\ \vdash\ e : T_2 \quad T_1 \sim T_1'}{\Gamma \vdash\ (\lambda x : T_1.e) : T_1 \rightarrow T_2}$$

Then the program $(\lambda x : \texttt{Int}.x\ x)$ is typeable because T_1' can be instantiated with $\star$. This program is static and untypeable in the original calculus STLC, therefore Criteria (1) is violated.

3.3 Step 3: Resolution of inputs

> $\mathbb{T}^c \overset{\texttt{toRes}}{\Longrightarrow} \mathbb{T}^r$: Inputs that do not have an occurrence as type-input are replaced by the join type of their outputs that have been considered up-to consistency.

Inputs receive their values from outputs. However, after step $\overset{\texttt{toCnst}}{\Longrightarrow}$ many outputs have been replaced by distinct variables. From which should an input receive from? In our methodology, we have a default choice and an exception. The default treatment is with the use of join types, as we can see for the case statement.

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : X \\ =_+ \ X\ T_1'\ T_2' \\ \Gamma, x : T_1 \vdash e_2 : T' \\ \Gamma, x : T_2 \vdash e_3 : T'' \\ T' \sqcap T'' = T^j\end{array}}{\Gamma \vdash (case\ e_1\ e_2\ e_3) : T} \overset{\texttt{toCnst}}{\Longrightarrow} \frac{\begin{array}{c}\Gamma \vdash e_1 : X \\ =_+ \ X\ T_1'\ T_2' \\ \Gamma, x : T_1' \vdash e_2\ T' \\ \Gamma, x : T_2' \vdash e_3\ T'' \\ T' \sqcap T'' = T^j\end{array}}{\Gamma \vdash (case\ e_1\ e_2\ e_3) : T^j}$$

Consider now the fix operator for general recursion.

$$\frac{\Gamma \vdash \ e : (T \to T)}{\Gamma \vdash \ (fix\ e) : T} \overset{\texttt{toCnst}}{\Longrightarrow} \frac{\begin{array}{c}\Gamma \vdash \ e : X \\ =_\to \ X\ T\ T' \quad T \sim T'\end{array}}{\Gamma \vdash \ (fix\ e : X) : T}$$

The presence of a type-input overrides the use of join types. Our methodology is such that if the language designer expands an output and reveal a type-input, then this latter is to be considered the reference for all types, that is, inputs points to it (simply by not being substituted and keeping the same name) and outputs will flow into it in the step for cast insertion. In the rule above, the input occurrence of T in the conclusion has not been replaced.

As the next step $\overset{\texttt{toSt}}{\Longrightarrow}$ is ineffectful for this example, this will be the final gradual type system. Notice that $<:$ must be such that $\star <: \star$. Moreover, we notice that our treatment follows the one of gradual typing and subtyping in [10] for objects (page 7, before the use of consistency-subtyping for avoiding non-determinism).

Violating our discipline

Example: Type-input outputs are not unique. Assume an application operator that takes two functions with a same signature and an argument and returns the pair of the results.

$$\frac{\begin{array}{c}\Gamma \vdash_G \ e_1 : T_1 \to T_2 \\ \Gamma \vdash_G \ e_2 : T_1 \to T_2 \\ \Gamma \vdash_G \ e_3 : T_1\end{array}}{\Gamma \vdash_G \ (e_1\ e_2\ e_3) : T_2 \times T_2}$$

Without showing the gradual type system, we can see that if type equality is left for T_1 then the program $((\lambda x : \texttt{Int}.x)\ (\lambda y : \star.y)\ 4)$ is not typeable. However, its more-precise program $((\lambda x : \texttt{Int}.x)\ (\lambda y : \texttt{Int}.y)\ 4)$ is typeable and therefore Criteria (2) is violated. This happens because type-inputs outputs still are outputs and should be separated. However, having multiple type-inputs makes it unclear what final types inputs should have and we have ruled out this possibility altogether.

3.4 Step 4: Ensuring staticity for free inputs

$\mathbb{T}^r \overset{\texttt{toSt}}{\Longrightarrow} \mathbb{T}^s$: Enforces that input types that are free, i.e., they do not receive a value from any output, will be static in the gradual type system.

We achieve this with premise $static(T)$ for all those inputs T that are free. The typing rule above for abstraction of ITLC is therefore translated in the following way.

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

It has been shown in [11] that violating this discipline fails in affording Criteria (1).

3.5 Compilation to the Cast Calculus

As the encoding term in the conclusion is in input position, output variables now are inputs and must be resolved. Then, the encoding variables are first cast to the expansion of their type (if their type is a pattern-matching variable), and then cast to a same type that uses the final type for those outputs that are not type-inputs.

Consider the $isnil$ function for lists and ignore for a moment the premises of the following rules.

$$\frac{\ldots}{\Gamma \vdash \ isnil[T]\ e : Bool} \overset{\texttt{toCI}}{\Longrightarrow} \frac{\ldots}{\Gamma \vdash_{CC} \ isnil[T]\ e \hookrightarrow isnil[T]\ e' : Bool}$$

In the conclusion of the rule that defines the cast insertion, the occurrence of T in the term $isnil[T]\ e'$, once an output, now finds itself in input position. This situation implies that the provability dynamics of the rule are such that the value for this occurrence is not provided by computation but by the rule itself. As this variable might appear in the context of a join, or might have a type-input of reference, it would be incorrect to receive from the original output (that is now among the premises of the rule). These variables, just as much as the inputs of Section 3.3, need to be resolved by $\overset{\texttt{toRes}}{\Longrightarrow}$.

The first cast to apply is for pattern-matching variables. Complex types can be nested at will and the cast must be directed to the full expansion of the type. For instance, here below we have a partial rule of the cast insertion for the try operator of exceptions. The term e_2' is of type X and generates the following cast. We show only the relevant term in the conclusion.

$$\frac{\begin{array}{c}\hookrightarrow e_1' : T' \qquad \hookrightarrow e_2' : X \\ =_\to \ X\ Y\ T'' \\ =_{\texttt{ExcType}} \ Y \\ T' \sqcap T'' = T^j\end{array}}{\hookrightarrow (e_2' : X \Rightarrow^{l_1} \texttt{ExcType} \to T'')}$$

Notice that T'' should not be considered the final destination. Indeed, it is involved in the calculation of a join. After this step there might be outputs that points to a *final type*: a join type or their type-input of reference. This introduces us to the final step of the methodology: we need to wrap this cast term with another cast that takes those types to their final. Applied to the rule above, this means that we further cast the term with $\texttt{ExcType} \to T'' \Rightarrow^{l_2} \texttt{ExcType} \to T^j$.

4. Gradualizing type systems

We have applied our methodology to nearly all the type systems of Pierce [7]. Namely, we have addressed STLC, unit type, pairs, tuples, records, let binding, general recursion (fix), sum types, exceptions, references, lists and STLC with subtyping. Figure 6 shows a handful of them and their relevant rules.

5. Typing systems in λ-prolog

Armed with the methodology described in Section 3, we now proceed to develop automatic procedures for manipulating type systems in the same way. We model type systems as logic programs in λ-prolog[2] [6]. Because of their rule-based nature, type systems naturally maps into logic programs. We make use of a few features of λ-prolog that suits the modeling of type systems for languages with binders (such as λ-calculi): types, higher order abstract syntax and hypothetical reasoning.

[2] In this paper we will be slightly inaccurate and always refer to λ-prolog as our working language. The reader should bear in mind however that λ-prolog is a concrete programming language (showed in Figure 2). Our theory makes use of the abstract logic behind λ-prolog that is the Harrop-hereditary logic, see [6].

Sum types

$$\frac{\Gamma \vdash_G e : T_1}{\Gamma \vdash_G (inl\ e : T_2) : T_1 + T_2} \quad \overset{\texttt{toGr}}{\Longrightarrow} \quad \frac{\begin{array}{c}\Gamma \vdash_G e : T_1' \\ (\sqcap T_1' = T_1') \\ (\sqcap T_2' = T_2')\end{array}}{\Gamma \vdash_G (inl\ e : T_2') : T_1' + T_2'} \quad \overset{\texttt{toCI}}{\Longrightarrow} \quad \hookrightarrow (inl\ e'\ T_2')$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : (T_1 + T_2) \\ \Gamma, x : T_1 \vdash e_2 : T \\ \Gamma, x : T_2 \vdash e_3 : T\end{array}}{\Gamma \vdash (case\ e_1\ e_2\ e_3) : T} \quad \overset{\texttt{toGr}}{\Longrightarrow} \quad \frac{\begin{array}{c}\Gamma \vdash e_1 : X \\ =_+\ X\ T_1'\ T_2' \\ \Gamma, x : T_1' \vdash e_2 : T' \\ \Gamma, x : T_2' \vdash e_3 : T'' \\ T' \sqcap T'' = T^j\end{array}}{\Gamma \vdash (case\ e_1\ e_2\ e_3) : T^j} \quad \overset{\texttt{toCI}}{\Longrightarrow} \quad \begin{array}{l}\hookrightarrow (case\ e_1' : X \Rightarrow^{l_1} T_1 + T_2 \\ \quad e_2' \colon T' \Rightarrow^{l_2} T^j\ e_3' \colon T'' \Rightarrow^{l_3} T^j)\end{array}$$

General recursion

$$\frac{\Gamma \vdash e : (T \to T)}{\Gamma \vdash (fix\ e) : T} \quad \overset{\texttt{toGr}}{\Longrightarrow} \quad \frac{\begin{array}{c}\Gamma \vdash e : X \\ =_\to\ X\ T\ T' \qquad T \sim T'\end{array}}{\Gamma \vdash (fix\ e : X) : T} \quad \overset{\texttt{toCI}}{\Longrightarrow} \quad \hookrightarrow fix\ (e : X \Rightarrow^{l_1} T \to T' \Rightarrow^{l_2} T \to T)$$

Lists

$$\frac{\Gamma \vdash e : list\ T}{\Gamma \vdash head[T] : T} \quad \overset{\texttt{toGr}}{\Longrightarrow} \quad \frac{\begin{array}{c}\Gamma \vdash e : X \\ =_{list}\ X\ T'' \\ T' \sqcap T'' = T^j\end{array}}{\Gamma \vdash head[T']\ e : T^j} \quad \overset{\texttt{toCI}}{\Longrightarrow} \quad \hookrightarrow head[T^j]\ (e' : X \Rightarrow^{l_1} list\ T' \Rightarrow^{l_2} list\ T^j)$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : list\ T \\ \Gamma \vdash e_2 : list\ T\end{array}}{\Gamma \vdash cons[T]\ e_1\ e_2 : list\ T} \quad \overset{\texttt{toGr}}{\Longrightarrow} \quad \frac{\begin{array}{c}\Gamma \vdash e_1 : X_1 \\ =_{list}\ X_1 T_2 \\ \Gamma \vdash e_2 : X_2 \\ =_{list}\ X_2 T_3 \\ T_1 \sqcap T_2 \sqcap T_3 = T^j\end{array}}{\Gamma \vdash cons[T_1]\ e_1\ e_2 : list\ T^j} \quad \overset{\texttt{toCI}}{\Longrightarrow} \quad \begin{array}{c}\hookrightarrow cons[T^j] \\ e_1' : X \Rightarrow^{l_1} list\ T_2 \Rightarrow^{l_2} list\ T^j \\ e_2' : X \Rightarrow^{l_1} list\ T_3 \Rightarrow^{l_2} list\ T^j\end{array}$$

References

$$\frac{\Gamma \vdash e : Ref\ T}{\Gamma \vdash\ !e : T} \quad \overset{\texttt{toGr}}{\Longrightarrow} \quad \frac{\begin{array}{c}\Gamma \vdash e : X \\ =_{Ref}\ X\ T\end{array}}{\Gamma \vdash\ !e : T} \quad \overset{\texttt{toCI}}{\Longrightarrow} \quad \hookrightarrow !(e' : X \Rightarrow^{l} Ref\ T)$$

$$\frac{\Gamma \vdash e_1 : Ref\ T \qquad \vdash e_2 : T}{\Gamma \vdash e_1 := e_2 : unit} \quad \overset{\texttt{toGr}}{\Longrightarrow} \quad \frac{\begin{array}{c}\Gamma \vdash e_1 : X \\ =_{Ref}\ X\ T \\ \vdash e_2 : T' \qquad T \sim T'\end{array}}{\Gamma \vdash e_1 := e_2 : unit} \quad \overset{\texttt{toCI}}{\Longrightarrow} \quad \hookrightarrow (e_1' : X \Rightarrow^{l_1} Ref\ T := e_2' : T' \Rightarrow^{l_2} T)$$

Exceptions

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : T \\ \Gamma \vdash e_2 : ExcType \to T\end{array}}{\Gamma \vdash try\ e_1\ with\ e_2 : T} \quad \overset{\texttt{toGr}}{\Longrightarrow} \quad \frac{\begin{array}{c}\Gamma \vdash e_1 : T' \qquad \Gamma \vdash e_2 : X \\ =_\to\ X\ Y\ T'' \\ =_{ExcType}\ Y \\ T' \sqcap T'' = T^j\end{array}}{\Gamma \vdash try\ e_1\ with\ e_2 : T^j} \quad \overset{\texttt{toCI}}{\Longrightarrow} \quad \begin{array}{c}\hookrightarrow try\ e_1' : T' \Rightarrow^{l_1} T^j\ with \\ e_2' : X \Rightarrow^{l_1} ExcType \to T'' \Rightarrow^{l_2} ExcType \to T^j\end{array}$$

STLC with algorithmic subtyping

$$\frac{\begin{array}{c}\Gamma \vdash_G e_1 : T_1 \to T_2 \\ \Gamma \vdash_G e_2 : T_3 \qquad T_3 <: T_1\end{array}}{\Gamma \vdash_G (e_1\ e_2) : T_2} \quad \overset{\texttt{toGr}}{\Longrightarrow} \quad \frac{\begin{array}{c}\Gamma \vdash_G e_1 : X \qquad =_\to\ X\ T_1\ T_2' \\ \Gamma \vdash_G e_2 : T_3' \qquad T_3' <: T_1' \\ T_1' \sim T_1\end{array}}{\Gamma \vdash_G (e_1\ e_2) : T_2} \quad \overset{\texttt{toCI}}{\Longrightarrow} \quad \hookrightarrow (e_1' : X \Rightarrow^{l_1} T_1 \to T_2\ e_2')$$

Figure 6. Examples of gradualized rules.

Typed logic programming Logic programs of λ-prolog must follow a typing discipline that is declared in their signature. The signature first defines the entities that are involved in the program. For instance, for STLC we would have terms and types and the signature contains the declaration `kind term` and `kind type`[3]. For brevity, we will assume that kind declarations are always present for the entities that are mentioned in programs. The real typing part is realized with declarations such as the following.

$$\begin{array}{ll} \texttt{int} : \texttt{type} & \texttt{bool} : \texttt{type} \\ \texttt{arrow} : \texttt{type} \rightarrow \texttt{type} \rightarrow \texttt{type} & \\ \texttt{app} : \texttt{term} \rightarrow \texttt{term} \rightarrow \texttt{term} & \\ \vdash : \texttt{term} \rightarrow \texttt{type} \rightarrow \texttt{prop} & \end{array}$$

Thanks to these declarations, expressions such as $\vdash\ \texttt{T}\,\texttt{T}$, for a logical variable `T`, and (`app int int`) are not well-typed. The signature will drive the Gradualizer in its automatic procedures. For instance, the pattern-matching predicate that realizes $=_{\rightarrow}$ will be automatically generated over the inspection of the declaration for `arrow`, and so on for the other constructors.

Higher order abstract syntax (HOAS) HOAS is an approach to syntax in which the underlying logic can appeal to a native λ-calculus that will help modeling aspects related to binding of the object language being specified. Suppose for instance that we would like to add the abstraction to the code above, we would define the following constructor `abs`.

$\texttt{abs} : \texttt{type} \rightarrow (\texttt{term} \rightarrow \texttt{term}) \rightarrow \texttt{term}$
Examples of programs:
$\lambda x : T.x = (\texttt{abs T}\ \lambda\texttt{x.x})$
$\lambda x : (int \rightarrow int).\lambda y : int.(x\ y) =$
$(\texttt{abs}\ (\texttt{arrow int int})\ \lambda\texttt{x}.(\texttt{abs int}\ \lambda\texttt{y}.(\texttt{app x y})))$

Notice that the second argument of `abs` is an abstraction and is employed as such in the examples of programs above. The *terms* of logic programs are therefore those that can be constructed interleaving λ terms with logical variables (`X`) and the constructors (`f`) from the signature.

Definition 1 (Terms)**.**

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

Hypothetical reasoning In order to appreciate the role of hypothetical reasoning, let us consider the typing rule for abstraction in the original STLC and that of its definition in λ-prolog.

$$\frac{\Gamma, x : T_1 \vdash\ e\ T_2}{\Gamma \vdash\ \lambda x : T_1.e\ T_1 \rightarrow T_2} \rightsquigarrow \frac{(\forall x.\vdash\ x\ T_1 \Rightarrow \vdash\ (R\ x)\ T_2)}{\vdash\ (\texttt{abs}\ T_1\ R)\ (\texttt{arrow}\ T_1\ T_2)}$$

What is implicit in the rule on the left is an appeal to the same typing relation $\vdash$ for a generic variable x. The premise of this rule reads "suppose that $\vdash\ x\ T_1$, can we prove $\vdash\ e\ T_2$?". In λ-prolog (on the right), this is made explicit with the use of generic reasoning for $\forall$ and hypothetical reasoning due to the implication $\Rightarrow$. Operationally speaking, encountering a goal with $\forall x$ will create a new fresh constant, and proving the implication above will temporarily augment the logic program with the fact $\vdash x\ T_1$, in order to prove the goal $\vdash\ e\ T_2$ in this augmented logic program. Whether the goal succeeds or not, the logic program will come back to the original one (without the new constant and the fact $\vdash x\ T_1$).

In logic programming, *formulae* are the main tool for expressing meaning. In order to graphically match our intuition we will still present formulae in the premises/conclusion style. Moreover, to simplify the Gradualizer, we shall use only a fragment of λ-prolog. Formulae and rules will have the following shape.

Definition 2 (Formulae, premises and rules)**.**

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

A rule is of the form

$$\frac{\{premise_i \mid i \in I\}}{formula}$$

where I is an indexing set. Given a rule r, $premises(r)$ denotes its set of premises and $conclusion(r)$ denotes the formula in its conclusion.

We now have all the ingredients for defining our notion of type system. Viewed in the setting of λ-prolog, type systems are simply logic for which is ensured that the signature contains the kinds `term` and `type` and the typability predicate.

Definition 3 (Type System)**.** *A* type system *is a triple* $(\Sigma, D, \vdash)$ *where Σ is a signature and D is a set of rules over Σ. Moreover, Σ contains the declaration* $\vdash : \texttt{term} \rightarrow \texttt{type} \rightarrow \texttt{prop}$.

The semantics of type systems is that of logic programs of λ-prolog [6]. In particular, the semantics is based on provability of formulae using the application of the rules of the type system. The encountering of universal quantification and implication is treated as described above, and the usage of HOAS behaves as expected. We will use the notation[4] $\mathbb{T} \models \Phi$ to denote that the formula Φ is provable by the type system $\mathbb{T}$.

6. The Gradualizer

We will assume that type systems in input adhere to the following restrictions: 1) all the terms of type `term` that appear in premises are logical variables from the conclusion and are used only once in the premises, 2) rules do not explicit use lambda abstraction 3) hypothetical formulae are of the form: $\forall x.(\vdash\ x\ t \Rightarrow\ \vdash\ t\ t)$ and 4) signatures cannot use nested abstractions and they can appear only with the form $(\texttt{term} \rightarrow \texttt{term})$ (in a sense, the use that `abs` make of HOAS).

The role of these restrictions is, admittedly, to simplify the procedures while retaining a large expressiveness. Indeed, all the examples considered in this paper adhere to these restrictions and we have applied the Gradualizer to such systems.

The procedures that follow make use of a few global functions. We assume that $\texttt{outputs}(T, r)$ and $\texttt{typeinput}(T, r)$ are true when the type T occurs in output and type-input position, respectively, in the rule r. The function $\texttt{sig}(pred, k)$ returns the kind of the k-th argument of $pred$, for instance $\texttt{sig}(\vdash, 1) = \texttt{term}$ and $\texttt{sig}(\vdash, 2) = \texttt{type}$. When we will apply a function with the notation $\bullet^{-1}$ will mean that the function will employ $\texttt{mode}^{-1}$ and $\texttt{capab}^{-1}$ in lieu of `mode` and `capab`.

Similarly to Section 3, each of the following subsections is devoted to one step of our methodology.

6.1 Step 1: Pattern-matching of outputs

We first generate the pattern-matching predicates by inspecting the declarations in the signature.

Definition 4 (Type Systems with Pattern Matching)**.** *A type system* $\mathbb{T} = (\Sigma, D, \vdash)$ extends a type system $\mathbb{T}' = (\Sigma', D', \vdash)$ with

[3] In the concrete syntax of λ-prolog, *type* is a reserved word and our implementation of the Gradualizer uses `kind typ`.

[4] We are aware that the symbol $\models$ is typical for semantic/model-theoretic satisfiability rather than proof system provability. However, the standard symbol $\vdash$ is also standard for typability.

pattern matching *whenever* Σ *extends* Σ' *and for all declarations* $f : T_1 \to \ldots \to T_n \to \texttt{type}$ *in* Σ', *with* $n \geq 0$ *it holds that*

- Σ *contains:* $=_f \colon \texttt{type} \to T_1 \to \ldots \to T_n \to \texttt{prop}$.
- D *contains the rules*

$$=_f \ (f\, X_1, \ \ldots \ , X_n)\, X_1 \ \ldots \ X_n.$$
$$=_f \ \underbrace{\star\ \star\ \ldots\ \star}_{(n+1)\ times}.$$

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

The transformation $\stackrel{\texttt{toPM}}{\Longrightarrow}$ is defined below. We have highlighted the salient parts.

Definition 5 ($\mathbb{T} \stackrel{\texttt{toPM}}{\Longrightarrow} \mathbb{T}^p$)**.** *Given type systems* $\mathbb{T}^p = (\Sigma, D, \vdash)$ *and* $\mathbb{T} = (\Sigma', D', \vdash)$, *we write* $\mathbb{T} \stackrel{\texttt{toPM}}{\Longrightarrow} \mathbb{T}^p$ *whenever* $\mathbb{T}^p$ *extends* $\mathbb{T}$ *with pattern-matching,* Σ *contains the declaration* $\star : \texttt{type}$ *and* D *is the least set such that for all rules* r *in* D', D *contains a rule* r' *such that*

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

  where $new\&pm$ *is defined as follows:*

$new\&pm(\Phi_1 \Rightarrow \Phi_2) = new\&pm^{-1}(\Phi_1) \Rightarrow new\&pm(\Phi_2)$
$new\&pm((pred\, t_1, \ \ldots, \ t_n)) = (pred\, t_1^*, \ \ldots, \ t_n^*)$
with $t_k^* = new\&pm(t_k)$,
if $\texttt{mode}(pred, k) = o$ *and* $\texttt{sig}(pred, k) = \texttt{type}$.
$t_k^* = t_k$, *otherwise.*
$new\&pm(f\ t_1\ \ldots\ t_n) = X$, *with* X *fresh in* r'
and $premise(r')$ *contains*
$=_f\ X\ new\&pm(t_1)\ \ldots\ new\&pm(t_n)$.
$new\&pm(t) = t$, *otherwise.*

Theorem 6 ($\mathbb{T}$ and $\mathbb{T}^p$ coincide)**.** *Given two type systems* $\mathbb{T} = (\Sigma, D, \vdash)$ *and* $\mathbb{T}^p$, *if* $\mathbb{T} \stackrel{\texttt{toPM}}{\Longrightarrow} \mathbb{T}^p$ *then for all static* e *and* T *of* $\mathbb{T}$, *it holds that*

$$\mathbb{T} \models \vdash\ e\, T \text{ if and only if } \mathbb{T}^p \models \vdash\ e\, T.$$

Proof. Both directions of the implications can be proved with an induction on the provability of $\mathbb{T}$ and $\mathbb{T}^p$, respectively.

6.2 Step 2: Up-to consistency for outputs

The transformation $\stackrel{\texttt{toCnst}}{\Longrightarrow}$ is done in two steps. The first step transforms the type system $\mathbb{T}$ into an equivalent one $\mathbb{T}^e$ where same type-output outputs are given new fresh variables. Moreover, a series of binary equalities are created in order to keep track of these equated types. The second step simply transforms $\mathbb{T}^e$ into a type system $\mathbb{T}^c$ that replaces all the equality premises for a variable with a single join premise. In order to do so, we need to detect all such equality premises (in Definition 7). The details for this part follows.

Definition 7 (Type Systems with Type Equality)**.** *A type system* $\mathbb{T} = (\Sigma, D, \vdash)$ extends a type system $\mathbb{T}' = (\Sigma', D', \vdash)$ with type equality *whenever* Σ *extends* Σ',
Σ *contains the declaration* $= \colon \texttt{type} \to \texttt{type} \to \texttt{prop}$, *and*
D *contains the rule* $=\ X\ X$, *for a logical variable* X.
Moreover, we say that P *is an* equationally maximal set of equalities for a rule r *of* $\mathbb{T}$ *whenever*

$$P \text{ is } \{=\ X\ X_1,\ =\ X\ X_2,\ \ldots\ , =\ X\ X_k\}$$

for some $k \geq 1$ *and variables* X, X_1, X_2, ..., X_k, *and for all premises* Φ *in* $premises(r)$ *it holds that either* $\Phi \in P$ *or* Φ *is not* $=\ X\ Y$, *for any variable* Y.

Given P *of the form above, we also define* $dom(P) = X$ *and* $cod(P) = \{X_1, \ \ldots \ , X_k\}$, *that is, with* X *excluded.*

Notice that the step $\stackrel{\texttt{toCnst}}{\Longrightarrow}$ is preparatory for the resolutions of inputs. After giving the join premises we also generate a consistency premise. Internally, the Gradualizer consider the consistency check as *directed*. If T is a type-input, then the join of its outputs should converge to it and we will have a premise $T \sim T^j$. The meaning of this premise is that T and T^j are consistent and also that T flows into T^j. If T is not a type-input, then all the occurrences of T are to be replaced by T^j, hence we use a swapped premise $T^j \sim T$. This difference is only operational to the Gradualizer, provability coincides for these two premises.

Definition 8 ($\mathbb{T} \stackrel{\texttt{toCnst}}{\Longrightarrow} \mathbb{T}^c$)**.** *Given type systems* $\mathbb{T}^c = (\Sigma, D, \vdash)$ *and* $\mathbb{T} = (\Sigma', D', \vdash)$, *we say* $\mathbb{T} \stackrel{\texttt{toCnst}}{\Longrightarrow} \mathbb{T}^c$ *whenever*
1) It exists a type system $\mathbb{T}^e = (\Sigma, D^e, \vdash)$ *that extends* $\mathbb{T}$ *with type equality such that for all rules* r *in* D', D^e *contains a rule* r' *such that*

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

2) Σ *extends* Σ' *and* D *is the least set of rules such that for all rules* r *in* D^e, D *contains a rule* r' *such that*

- $conclusion(r') = conclusion(r)$, *and*
- $premises(r')$ *is the least set such that*
  - *for all premises* Φ *of* r *that are not type equality,* $premises(r')$ *contains the premise* Φ
  - *for all equationally maximal set of equalities* P *for* r *with cardinality* k, $\mathbb{T}^c$ *extends* $\mathbb{T}$ *with the* k*-ary join and consistency and* r' *contains the premises* $joinAndFlow(P, r)$.

$joinAndFlow(P, r) = \bigsqcap cod(P) = X^j$ *and also either*
$X^j \sim dom(P)$, *if* $\texttt{typeinput}(dom(P), r)$, *or*
$dom(P) \sim X^j$, *otherwise.*

$r\&eq(\Phi_1 \Rightarrow \Phi_2) = r\&eq^{-1}(\Phi_1) \Rightarrow r\&eq(\Phi_2)$
$r\&eq((pred\, t_1, \ \ldots, \ t_n)) = (pred\, t_1^*, \ \ldots, \ t_n^*)$
with $t_k^* = r\&eq(t_k)$,
if $\texttt{mode}(pred, k) = o$ *and* $\texttt{sig}(pred, k) = type$.
$t_k^* = t_k$ *otherwise.*
$r\&eq(f\ t_1\ \ldots\ t_n) = (f\ r\&eq(t_1)\ \ldots\ r\&eq(t_n))$.
$r\&eq(T) = T'$, *for some variable* T' *fresh in* r', *and*
$premises(r')$ *contains* $=\ T\ T'$.
$r\&eq(T) = T$ *if* $\texttt{typeinput}(T, r)$, *otherwise.*

At the second step, join and consistency premises are added and equalities are not copied into $\mathbb{T}^c$ which, in fact, contains no equality premises.

Theorem 9 ($\mathbb{T}$ and $\mathbb{T}^c$ coincide)**.** *Given the type systems* $\mathbb{T} = (\Sigma, D, \vdash)$ *and* $\mathbb{T}^c$, *if* $\mathbb{T} \stackrel{\texttt{toCnst}}{\Longrightarrow} \mathbb{T}^c$ *then for all static* e *and* T *of* $\mathbb{T}$, *it holds that*

$$\mathbb{T} \models \vdash\ e\, T \text{ if and only if } \mathbb{T}^c \models \vdash\ e\, T.$$

Proof. We first prove the statement for $\mathbb{T}$ and the type system $\mathbb{T}^e$ of the first step of Definition 8. We then prove the statement for $\mathbb{T}^e$ and $\mathbb{T}^c$ by induction on their provability.

6.3 Step 3: Resolution of inputs

It is convenient to define a notion of *final types* for an input type, i.e. the type from which the input will receive its value from. Thanks to our preparatory work in the previous step, we will have consistency premises to simply define final types.

Definition 10 (Final type of an input variable)**.** *Given a variable T and a rule r, we define* the final type of T, written T^{Fi}, *as*
$T^{Fi} = T'$, *if* $premises(r)$ *contains* $T \sim T'$.
$T^{Fi} = T$, *otherwise.*

The final type transformation substitutes only input variables. However, variables in pattern-matching premises such as $=_{Bool}\ X$ will not be affected because the previous step does not generate consistency premises for them. The definition of $\stackrel{\texttt{toRes}}{\Longrightarrow}$ is the following.

Definition 11 ($\mathbb{T} \stackrel{\texttt{toRes}}{\Longrightarrow} \mathbb{T}^r$)**.** *Given type systems* $\mathbb{T}^r = (\Sigma, D, \vdash)$ *and* $\mathbb{T} = (\Sigma', D', \vdash)$*, we say* $\mathbb{T} \stackrel{\texttt{toRes}}{\Longrightarrow} \mathbb{T}^r$ *whenever* Σ *extends* Σ' *and* D *is the least set such that for all rules* r *in* D'*,* r *adheres to the flow restriction and* D *contains a rule* r' *such that*

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

where $j\&fl$ *is defined as follows:*

$j\&fl(\Phi_1 \Rightarrow \Phi_2) = j\&fl^{-1}(\Phi_1) \Rightarrow j\&fl(\Phi_2)$
$j\&fl((pred\ t_1,\ \ldots,\ t_n)) = (pred\ t_1^*,\ \ldots,\ t_n^*)$
with $t_k^* = t_k^{Fi}$,
  if $\texttt{mode}(pred, k) = i$ *and* $\texttt{sig}(pred, k) = type$,
  $t_k^* = t_k$, *otherwise.*

Theorem 12 ($\mathbb{T}$ and $\mathbb{T}^r$ coincide)**.** *Given type systems* $\mathbb{T} = (\Sigma, D, \vdash)$ *and* $\mathbb{T}^r$*, if* $\mathbb{T} \stackrel{\texttt{toRes}}{\Longrightarrow} \mathbb{T}^r$ *then for all static* e *and* T *of* $\mathbb{T}$*, it holds that*

$$\mathbb{T} \models \vdash\ e\,T \text{ if and only if } \mathbb{T}^r \models \vdash\ e\,T.$$

Proof. By induction on the provability of $\mathbb{T}$ and $\mathbb{T}^r$, respectivelty.

6.4 Step 4: Ensuring staticity for free inputs

The transformation $\stackrel{\texttt{toSt}}{\Longrightarrow}$ is rather simple.

Definition 13 ($\mathbb{T} \stackrel{\texttt{toSt}}{\Longrightarrow} \mathbb{T}^s$)**.** *Given type systems* $\mathbb{T}^s = (\Sigma, D, \vdash)$ *and* $\mathbb{T} = (\Sigma', D', \vdash)$*, we say* $\mathbb{T} \stackrel{\texttt{toSt}}{\Longrightarrow} \mathbb{T}^s$ *whenever* Σ *extends* Σ' *and* D *is the least set such that for all rules* r *in* D'*,* D *contains a rule* r' *such that*

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

where $f\&st$ *is defined as follows:*

$f\&st(\Phi_1 \Rightarrow \Phi_2) = f\&st^{-1}(\Phi_1) \Rightarrow f\&st(\Phi_2)$
$f\&st((pred\ t_1,\ \ldots,\ t_n)) = (pred\ t_1^*,\ \ldots,\ t_n^*)$
with $t_k^* = f\&st(t_k)$,
  if $\texttt{mode}(pred, k) = i$ *and* $\texttt{sig}(pred, k) = type$.
  $t_k^* = t_k$ *otherwise.*
$f\&st(f\ t_1\ \ldots\ t_n) = (f\ f\&st(t_1)\ \ldots\ f\&st(t_n))$.
$f\&st(T) = T$ *and if not* $\texttt{outputs}(T, r)$
  then $premises(r')$ *contains the premise* $static(T)$.

Theorem 14 ($\mathbb{T}$ and $\mathbb{T}^s$ coincide)**.** *Given the type systems* $\mathbb{T} = (\Sigma, D, \vdash)$ *and* $\mathbb{T}^s$*, if* $\mathbb{T} \stackrel{\texttt{toSt}}{\Longrightarrow} \mathbb{T}^s$ *then for all static* e *and* T *of* $\mathbb{T}$*, it holds that*

$$\mathbb{T} \models \vdash\ e\,T \text{ if and only if } \mathbb{T}^s \models \vdash\ e\,T.$$

Proof. By induction on the provability of $\mathbb{T}$ and $\mathbb{T}^s$, respectivelty.

6.5 Compilation to the Cast Calculus

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

Definition 15 (Type system of the Cast Calculus)**.** *A type system* $\mathbb{T}^{CC} = (\Sigma, D, \vdash_{CC})$ extends the type system $\mathbb{T} = (\Sigma', D', \vdash)$ with casts, *written* $\mathbb{T} \stackrel{\texttt{toCC}}{\Longrightarrow} \mathbb{T}^{CC}$*, whenever* $\mathbb{T}^{CC}$ *extends* $\mathbb{T}$*, and*

- Σ *contains the declarations*
  $\star : \texttt{type}$.
  $\vdash_{\hookrightarrow} : \texttt{term} \to \texttt{term} \to \texttt{type} \to \texttt{prop}$ *with mode* (i, o, o).
  $\langle\,\rangle : \texttt{type} \to \texttt{type} \to \texttt{label} \to \texttt{term}$.
- D *imports the rules for* $\vdash$ *from* D' *but named for the predicate* $\vdash_{CC}$*, and* D *contains also the rule*

$$\frac{\Gamma \vdash\ e : T_1}{\Gamma \vdash\ \ : T_1 \Rightarrow^l T_2 e : T_2}$$

The next step is to dive straight into the compilation. Thanks to Restriction 1, identifying the terms to cast in the conclusion is not problematic. As we have seen in Section 3.5, pattern-matching variables lead to a cast to their full type. To this aim, we develop a notion of of final types for these variables.

Definition 16 (Final type of pattern-matching variable)**.** *Given a variable* X *and a rule* r*, the* final type of X, written X^{Fpm} *is defined as follows.*

$X^{Fpm} = (f\ X_1^{Fpm}\ \ldots\ X_n^{Fpm})$,
  if $premises(r)$ *contains* $=_f\ X\ X_1\ \ldots\ X_n$.
$X^{Fpm} = X$*, otherwise.*

Also, outputs lead to a cast involving their type-input output of reference, if present, or their join type otherwise. We thus have the following notion of final type for output variables as well.

Definition 17 (Final type of output variables)**.** *Given a variable* X *and a rule* r*, the* final type of X, written X^{Fo} *is defined as follows.*

$T^{Fo} = T_1^{Fi}$ *if* $premises(r)$ *contains* $\bigsqcap \tilde{T} = T_1$ *and* $T \in \tilde{T}$
$T^{Fo} = T$*, otherwise.*

In what follows, we assume that the function enc, that stands for encoding, returns the same fresh new variable (that was not in the rule) for same inputs. We need this function because the encoding of a term e must be to a fresh new variable e' that appears both in the premises and in the conclusion.

As λ-prolog makes use of HOAS, the Gradualizer will 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 and our procedure generates a wrapped term $\lambda x.((E\ x) : T_1 \Rightarrow^l T_2)$.

Definition 18 (Cast Calculus with Compilation from the Gradual Calculus)**.** *Given the type systems* $\mathbb{T} = (\Sigma, D, \vdash)$*,* $\mathbb{T}^G = (\Sigma', D', \vdash_G)$ *and* $\mathbb{T}^{CC} = (\Sigma'', D'', \vdash_{CC})$ *such that* $\mathbb{T} \stackrel{\texttt{toGr}}{\Longrightarrow} \mathbb{T}^G$ *and* $\mathbb{T} \stackrel{\texttt{toCC}}{\Longrightarrow} \mathbb{T}^{CC}$*. We say that* $\mathbb{T}^G$ generates the cast insertion into $\mathbb{T}^{CC}$*, written* $\mathbb{T}^G \stackrel{\texttt{toCI}}{\Longrightarrow} \mathbb{T}^{CC}$*, whenever for all rules* r *in* D' *that define* $\vdash_G$*,* D'' *contains a rule* r' *such that*

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

where $enc\&cast$ *is defined as follows:*

$enc\&cast(\Phi_1 \Rightarrow \Phi_2) = enc\&cast^{-1}(\Phi_1) \Rightarrow enc\&cast(\Phi_2)$
$enc\&cast(\vdash_G \ e\ t) = (\Gamma \vdash_{CC} \ e \hookrightarrow e^* : t)$
with $e^* = enc(cast(e^{Fi}))$ *if* $\mathtt{mode}(pred, k) = i$.
$e^* = enc(e)$, *otherwise.*
$enc\&cast(\Phi) = \Phi$, *otherwise.*
$cast(e) = e\sigma$, *where* σ *is defined as follows:*
for all variables $E \in vars(e)$,

- *for all premises* $\vdash_G \ E\ X$ *in* r, $\sigma(E) = E : X \Rightarrow^{L_1} X^{Fpm} \Rightarrow^{L_2} X^{Fo}$.
- *for all premises* $\vdash_G \ (E\ x)\ X$ *in* r, $\sigma(E) = E^{Fpm}$, *where* $E^{Fo} = \lambda y.((E\ y) : X^{Fpm} \Rightarrow^{L_2} X^{Fo})$ $E^{Fpm} = \lambda z.((E^{Fo}\ z) : X \Rightarrow^{L_1} X^{Fpm})$

The substitution σ *is the identity everywhere else. In each case,* L *is a fresh variable in* r. *To avoid unnecessary casts,* $X \Rightarrow^{L_1} X^{Fpm}$ *is done only when* X *is a pattern-matching variable.*

7. Correctness of the Gradualizer

We have proved that the type systems produced by the Gradualizer always satisfy Criteria (1), i.e., typeability coincides over static terms.

Theorem 19. *Given two type systems* $\mathbb{T}$ *and* $\mathbb{T}^G$, *if* $\mathbb{T} \overset{\mathtt{toGr}}{\Longrightarrow} \mathbb{T}^G$ *then for all* e *and* T *of* $\mathbb{T}$, *it holds that*

$$\vdash \ e\,T \text{ if and only if} \vdash_G \ e\,T.$$

Proof. It is a straightforward consequence of the theorems that we have proved along the way, namely, Theorem 6, 9, 12 and 14.

In order to address Criteria (2), we need to generalize the less-precision relation $\sqsubseteq$ to our setting. In particular, we can derive it from the signature of the type system.

Definition 20 (Less-precision relations for a gradually typed system)**.** *Given a gradual type system* $\mathbb{T}^G = (\Sigma, D, \vdash_G)$, *we say that a set* $\{\sqsubseteq_S : S \times S \mid S \text{ is a type in } \Sigma\}$ *is* the set of less-precision relations for $\mathbb{T}^G$ with bottom $\star$ *whenever it holds that*

- *for all* $f : S_1 \to S_2 \ldots \to S_n \to S \in \Sigma$, *for some* $n \geq 0$,
$$\frac{t_1 \sqsubseteq_{S_1} t_1' \quad t_2 \sqsubseteq_{S_2} t_2' \quad \ldots \quad t_n \sqsubseteq_{S_n} t_n'}{(f\ t_1\ t_2\ \ldots\ t_n) \sqsubseteq_S (f\ t_1'\ t_2'\ \ldots\ t_n')}$$
- Σ *contains the declaration* $\star : \mathtt{type}$ *and it holds that* $\star \sqsubseteq_{type} t$ *for all terms* t *of type* $\mathtt{type}$.

We have then proved one of the major theorems of this paper: the Gradualizer produces only type systems that satisfy Criteria (2).

Theorem 21. *Given two type systems* $\mathbb{T}$ *and* $\mathbb{T}^G$, *if* $\mathbb{T} \overset{\mathtt{toGr}}{\Longrightarrow} \mathbb{T}^G$ *then it holds that for all* e, e' *and* T *of* $\mathbb{T}^G$,

$$\text{if} \vdash_G \ e\,T \text{ and } \ e' \sqsubseteq e \text{ then it exists } T' \text{ such that } \vdash_G \ e'\,T' \text{ and } T' \sqsubseteq T.$$

Proof. Induction on the provability of $\vdash_G$.

We can now turn to the cast calculus and prove Criteria (3).

Theorem 22 (Cast Insertion exists and is type preserving)**.** *Given the type systems* $\mathbb{T}$, $\mathbb{T}^G$ *and* $\mathbb{T}^{CC}$. *If* $\mathbb{T} \overset{\mathtt{toGr}}{\Longrightarrow} \mathbb{T}^G$, $\mathbb{T} \overset{\mathtt{toCC}}{\Longrightarrow} \mathbb{T}^{CC}$ *and* $\mathbb{T}^G \overset{\mathtt{toCI}}{\Longrightarrow} \mathbb{T}^{CC}$ *it holds that for all* e, e', *and* T *of* $\mathbb{T}^G$,

$$\text{if} \vdash_G \ e\,T \text{ then} \vdash_{CC} \ e \hookrightarrow e' : T \text{ and} \vdash_{CC} \ e'\,T.$$

Proof. Induction on the provability of $\vdash_{CC} \hookrightarrow$.

Another important criteria for the compilation is that it be monotonic w.r.t. the precision relation. We have not fully addressed this property for the Gradualizer. In fact, as the procedures depend on input/output information provided by the user we will need a deeper analysis of this aspect. Notice that this aspect does not occur for Criteria (3) because regardless of those choices the variables are substituted accordingly throughout the rule.

8. The implementation of the Gradualizer

We have implemented the Gradualizer and it can be downloaded at the github repository [3].This tool takes in input the implementation of a type system in λ-prolog and produces the typechecker and the cast insertion procedure to the cast calculus in λ-prolog. We have applied our tool to the type systems mentioned in Section 4 and the repository also contains the generated typecheckers and the compilers.

9. Conclusions

Overall, this paper is meant to serve as a helpful reference for those language designers who wants to lift their languages to gradual typing. In this regards, we believe that the methodology here put forward, the Gradualizer and its implementation will be essential tools for supporting and automating such a lift.

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] M. Cimini. Gradualizer. https://github.com/mcimini/Gradualizer, 2015.

[4] R. Garcia and M. Cimini. Principal type schemes for gradual programs. In *Symposium on Principles of Programming Languages*, POPL, pages 303–315, 2015. . URL `http://doi.acm.org/10.1145/2676726.2676992`.

[5] D. Herman, A. Tomb, and C. Flanagan. Space-efficient gradual typing. *Higher Order Symbol. Comput.*, 23 (2):167–189, June 2010. ISSN 1388-3690. . URL `http://dx.doi.org/10.1007/s10990-011-9066-z`.

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

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

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

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

[10] 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.

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

[12] J. G. Siek and P. Wadler. Threesomes, with and without blame. In *Proceedings for the 1st workshop on Script to Program Evolution*, STOP '09, pages 34–46, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-543-7. . URL `http://doi.acm.org/10.1145/1570506.1570511`.

[13] 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.

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