Putting Gradual Types to Work

Bhargav Shivkumar[1,2]($\boxtimes$) , Enrique Naudon[1] , and Lukasz Ziarek[2]

[1] Bloomberg, New York, USA
`bhargavs@buffalo.edu`
[2] SUNY - University at Buffalo, New York, USA

Abstract. In this paper, we describe our experience incorporating gradual types in a statically typed functional language with Hindley-Milner style type inference. Where most gradually typed systems aim to improve static checking in a dynamically typed language, we approach it from the opposite perspective and promote dynamic checking in a statically typed language. Our approach provides a glimpse into how languages like SML and OCaml might handle gradual typing. We discuss our implementation and challenges faced—specifically how gradual typing rules apply to our representation of composite and recursive types. We review the various implementations that add dynamic typing to a statically typed language in order to highlight the different ways of mixing static and dynamic typing and examine possible inspirations while maintaining the gradual nature of our type system. This paper also discusses our motivation for adding gradual types to our language, and the practical benefits of doing so in our industrial setting.

Keywords: Gradual typing · Type inference · Functional programming

1 Introduction

Static typing and dynamic typing are two opposing type system paradigms. Statically typed languages are able to catch more programmer bugs early in the compilation process, at the expense of a more flexible semantics. On the other hand, dynamically typed languages allow greater flexibility, while allowing more bugs at runtime. The proponents of each paradigm often feel very strongly in favor of their paradigm. Language designers are stranded in the middle of this dichotomy and left to decide between the two extremes when designing their languages.

At Bloomberg, we have felt this pain while designing a domain specific language for programmatically defining financial contracts. For the purposes of this paper, we will call our language Bloomberg Contract Language (BCL). BCL is a statically typed functional language with Hindley-Milner style type inference [4, 17], structural composite types and recursive types. Users of BCL are split into two groups—end users and language maintainers. End users are typically financial professionals whose primary programming experience involves scripting in dynamically typed languages such as Python and MATLAB. On the other hand, language maintainers are Bloomberg software engineers who are most at ease programming in statically typed and often functional languages like OCaml. Whilst it is of paramount importance to provide our end users with

an environment in which they are comfortable, our domain—financial contracts—is one in which correctness is of extraordinary importance, since errors can lead to large financial losses. This makes static types appealing, as they catch many errors that dynamic systems might miss. Even though static types provide a more error-free runtime, they do require extra effort from our end users who must learn an unfamiliar system. Our desire to simultaneously satisfy our end users and our language maintainers led us to gradual typing [23], which seeks to integrate static and dynamic typing in one system. Gradual typing in BCL allows language maintainers to stick to static typing and end users to selectively disable static typing when it interferes with their ability to work in BCL.

Since its introduction, gradual typing [23] has been making its way into more mainstream languages [29, 30] and more people have acknowledged the varied benefits of mixing static and dynamic typing in the same program. As identified by Siek and Taha [26], there has been considerable interest in integrating static and dynamic typing, both in academia and in industry. There has also been a plethora of proposed approaches, from adding a dynamic keyword [2], to using objects in object-oriented languages [16], to Seik and Taha's gradual typing itself [23]. While there seems to be no one-size-fits-all approach to designing a system that mixes static and dynamic types, Siek and Taha standardize the guarantees [26] we can expect from such a system. For language designers, this provides a more methodical way to approach the integration. Language designers can also draw from a large body of literature exploring the combination of gradual types with other common features, such as objects [24] and type inference [5, 25].

While it is typical for dynamically typed languages to go the gradual route in order to incorporate more static type checking, we go the other way and add more dynamism to our already static language. Most static languages that incorporate dynamic typing do so by following in the footsteps of Abadi et al. [2]–C# is a prime example of this [9]. Since BCL already supports type inference and we want to retain the dynamic feel of the language, we implement the inference algorithm described by Siek and Vachhrajani [25], putting us in an interesting position. Our approach promotes the use of a ? annotation to explicitly signify dynamically typed terms while un-annotated terms are (implicitly) statically typed, much like that of Garcia and Cimini [5]. This approach provides a simple escape hatch to end users who want to use dynamic typing as well as avenues to automate this process to ensure backwards compatibility of BCL with legacy code.

Finally, we feel there is a need to study the adaptation of gradual types to an existing language with a substantial user base and lots of existing code. We aim to provide a technical report in this paper that models our design decisions and implementation details of bringing in gradual types to BCL. Our primary contributions include:

– A brief review of other statically typed languages that add dynamic types, to compare and possibly derive inspiration for our own design in Sect. 2.
– Introduce a new use case that shows how a gradually typed language benefits different user groups of a language in Sect. 3.
– An inference algorithm, which is an adaptation of a prominent inference algorithm to add gradual types to a language with type inference in Sect. 4.

Note that throughout this paper we use "gradual" to indicate an implementation that provides gradual guarantees as specified in [26]. While, we do not state this formally for

BCL and leave that to future work, our implementation supports the smooth evolution of programs from static to dynamic typing as prescribed for gradually typed systems.

2   Background

In this section we briefly survey the existing literature to better contextualize our design choices. The incorporation of static and dynamic typing has been extensively studied [8,15,23,26,28], though usually in the context of a core calculus instead of a full-featured language. There also seems to be a juxtaposition of the literature, which generally follows a static-first approach, and practical implementations, which generally follow a dynamic-first approach[1] [7].

Abadi et al. [2] has been an inspiration for many static languages looking to incorporate dynamic typing. This work is a precursor to gradual typing, and while it does not qualify as gradual à la [23], it is nevertheless a standard when it comes to adding dynamic checks to a static language. Abadi's work uses a dynamic construct to build terms of type Dynamic and a typecase construct to perform case analysis on the runtime type of an expression of type Dynamic. This is similar to the typeof() function in dynamic languages like Python, which resolve the type of an expression at runtime. Siek and Taha observe that translating from their language of explicit casts to Abadi et al.'s language is not straightforward [23]. Nevertheless we believe that it is worthwhile to introduce something like the typecase construct in a static language with gradual types. We identify and discuss some potential applications of this in Sect. 5.

Statically typed object oriented languages like C# and Java have worked to incorporate some form of dynamic typing [6,16]. C# 4.0 introduced the dynamic type to declare objects that can bypass static type checking [1]. Although this achieves dynamic type checking, there is no indication of it being gradual à la [23]. Moreover, using the dynamic type in a C# program runs the program on the Dynamic Language Runtime (DLR) which is a separate runtime from the Common Language Runtime and which supports dynamic checking.

While works like [18,22] examine gradual type inference from the perspective of removing dynamic checks by performing type inference at runtime, Garcia and Cimini [5] (much like BCL) deals with static reasoning about programs, based on the consistency relation. [5] explores an alternate approach to gradual type inference and presents a statically typed language and its gradual counterpart. Instead of inferring gradual types based on type precision [25], this work limits the inference problem to static types only and requires consistency constraints between gradual types. An interesting feature of their language is that they distinguish between static type parameters and gradual type parameters to tell static parametric polymorphism apart from polymorphism due to the dynamic type.

Our approach is to adopt the properly gradual system defined by Siek and Vachchrajani [25]. That work describes the incorporation of gradual typing into a language with

---

[1] Here, static-first refers elaborating a static surface language to a gradually typed intermediate representation. Conversely, by dynamic-first we mean the opposite: elaborating a dynamic surface language to a gradually typed intermediate representation.

unification-based type inference. Unification-based inference is a common implementation of the Hindley-Milner type system [17], and is the implementation that BCL already uses. This makes our integration work relatively easier and also lets us leverage all the benefits of the standard for gradual typing laid out by Siek and Taha [26].

2.1 Gradual Types and Unification Based Inference

Siek and Vachchrajani [25] (S&V) propose an innovative solution for performing gradual type inference which combines gradual typing with type inference. Their main goal is to allow inference to operate on the statically typed parts of the code, while leaving the dynamic parts to runtime checks. Furthermore, the dynamic type must unify with static types and type variables, so that the static and dynamic portions of code may freely interact. In this section, we summarize their work.

The work of S&V is based on the gradually typed lambda calculus [23]. The gradually typed lambda calculus extends the simply typed lambda calculus ($\lambda_\rightarrow$) with an unknown type, ?–pronounced "dynamic"; type checking for terms of this type is left until runtime. The gradually typed lambda calculus ($\lambda_\rightarrow^?$) allows static and dynamic types to freely mix and satisfies the gradual guarantee [26], ensuring smooth migration between static and dynamic code while maintaining the correctness of the program.

(SVAR) $\dfrac{\Gamma(x) = \tau}{S; \Gamma \vdash x : \tau}$ $\boxed{S; \Gamma \vdash e : \tau}$ (GVAR) $\dfrac{\Gamma(x) = \tau}{S; \Gamma \vdash_g x : \tau}$ $\boxed{S; \Gamma \vdash_g e : \tau}$

(SCNST) $S; \Gamma \vdash c : typeof(c)$ (GCNST) $S; \Gamma \vdash_g c : typeof(c)$

(SAPP) $\dfrac{S; \Gamma \vdash e_1 : \tau_1 \quad S; \Gamma \vdash e_2 : \tau_2 \quad S(\tau_1) = S(\tau_2 \to \tau_3)}{S; \Gamma \vdash e_1\ e_2 : \tau_3}$ (GAPP) $\dfrac{S; \Gamma \vdash_g e_1 : \tau_1 \quad S; \Gamma \vdash_g e_2 : \tau_2 \quad S \models \tau_1 \simeq \tau_2 \to \beta \quad (\beta fresh)}{S; \Gamma \vdash_g e_1 e_2 : \beta}$

(SABS) $\dfrac{S; \Gamma(x \mapsto \tau_1) \vdash e : \tau_2}{S; \Gamma \vdash \lambda x : \tau_1.e : \tau_1 \to \tau_2}$ (GABS) $\dfrac{S; \Gamma(x \mapsto \tau_1) \vdash_g e : \tau_2}{S; \Gamma \vdash_g \lambda x : \tau_1.e : \tau_1 \to \tau_2}$

(a) $\lambda_\rightarrow^\alpha$ (b) $\lambda_\rightarrow^{?\alpha}$

Fig. 1. Simply and gradually typed lambda calculus with type variables
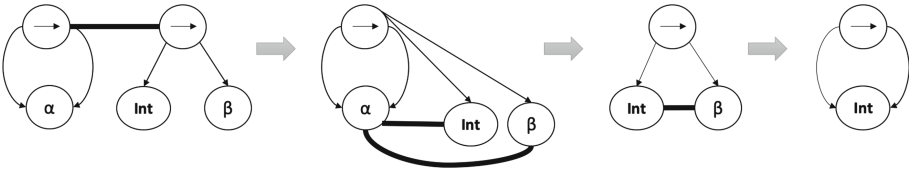
Fig. 2. Huet's unification of $\{\alpha \to \alpha = Int \to \beta\}$

Type inconsistencies in $\lambda_{\rightarrow}^{?}$ are caught by a *consistent* relation, instead of equality as in $\lambda_{\rightarrow}$. The *consistent* relation only compares parts of a type that are statically known; it is one of the key contributions of $\lambda_{\rightarrow}^{?}$. All type errors that cannot be statically resolved by the gradual type system are delegated to runtime checks.

Type inference allows programmers to omit type annotations in their programs and have the compiler infer the types for them. Hindley-Milner type inference is often cast as a two step process that consists of generating constraints and then solving them by a unification algorithm [20,21,32]. The inference algorithm models the typing rules as equations, called constraints, between type variables, while the unification algorithm computes a substitution S, which is a mapping from type variables to types, such that for each equation $\tau_1 = \tau_2$, we have $S(\tau_1) = S(\tau_2)$.

S&V introduce the gradually typed lambda calculus with type variables ($\lambda_{\rightarrow}^{?\alpha}$), which is $\lambda_{\rightarrow}^{?}$ extended with type variables, α. They define a new relation, *consistent-equal* ($\simeq$), which extends the *consistent* relation from $\lambda_{\rightarrow}^{?}$ to treatment α. Figure 1b compares the typing rules for $\lambda_{\rightarrow}^{\alpha}$, the statically typed lambda calculus with type variables, to the new type system $\lambda_{\rightarrow}^{?\alpha}$. S&V also specify a unification algorithm for $\lambda_{\rightarrow}^{?\alpha}$ which integrates the *consistent-equal* into Huet's unification algorithm [10,14] which is a popular algorithm that doesn't rely on substitution.

Huet's unification algorithm uses a graph representation for types. For example, a type like $Int \rightarrow \beta$ is represented as a sub graph in Fig. 2. A node represents a type, ground types, type variables or the function type ($\rightarrow$), and edges connect the nodes of types belonging to a $\rightarrow$ type. From this it follows that the unification algorithm is the amalgamation of two graphs present in a constraint equation following the rules of the type system. Huet's algorithm maintains a union find structure [27] to maintain equivalence classes among nodes and thereby types. When node A unifies with node B according to the type rules, the merge results in one of the two nodes becoming the representative of the merge. This signifies that the representative node is the solution to the constraint being unified. Figure 2 shows how the unification of the constraint $\{\alpha \rightarrow \alpha = Int \rightarrow \beta\}$ proceeds.

3   Introduction to BCL

Our motivation to explore gradual types for BCL is rooted in several historical and contextual details, which we discuss in this section. It is first helpful to understand that BCL is predominantly used to model financial contracts, by providing end users with programmatic access to a financial contract library. The library we use is based upon the composable contracts of Peyton Jones, Eber and Seward [19]. Its internal contract data structure is used throughout our broader derivatives system to support various downstream analyses. In this way, BCL serves as an expressive front-end for describing contracts to our derivatives system.

Let us look at a short illustrative example of BCL code. Figure 3 provides an example of the sort of thing for which BCL might be used. The `european_stock_option` function produces a `Contract` which models a European stock option. European stock options grant their holder the right, but not the obligation, to buy or sell stock in a company. The "European" in European stock option

```
let receive currency amount = scale (one currency) amount in

let european_stock_option args =
  let first = stock_price args.effective_date args.company in
  let last = stock_price args.expiry_date args.company in
  let payoff = match args.call_or_put with
    | Call -> (last / first - args.strike)
    | Put -> (args.strike - last / first)
  in
  european args.expiry_date (receive args.currency payoff)
in

european_stock_option
  { company = "ABC Co.",
    call_or_put = Call,
    strike = 100.0,
    currency = USD,
    effective_date = 2021-01-17,
    expiry_date = 2021-01-22 }
```

Fig. 3. European stock option

refers to the fact that, on one specific date, the holder must choose whether or not s/he would like to buy (or sell) the stock. This is in contrast to "American" options, where the holder may choose to buy (or sell) on any date within a specified range of dates.

This stock option is based on several helper functions, defined in [19], which we must examine first. The european function constructs a contract which allows its holder to choose between receiving "something" or nothing on a specified date. receive constructs a contract that pays the specified amount of the specified currency passed as arguments and uses the scale and one primitives. The scale primitive takes an amount of type *Obs Double*–where type *Obs d* represents a time-varying quantity of type *d*–and a contract as arguments and multiplies key values in the contract by the amount. Note that european_stock_option uses – and / operators which are built-ins that operate on *Obs Double* arguments. stock_price is a primitive for looking up the price of the specified stock on the specified date.

european_stock_option starts off by using stock_price to look up the price of the specified company's stock on the "effective" (contract start) and "expiry" (contract end) dates. It uses these stock prices to construct the payoff based on the specified call or put style, and feeds the payoff to receive to construct a contract that pays it. Finally european_stock_option passes the result of receive to the european, which allows the holder to choose between the payoff and nothing. Note that the payoff may well be negative, so the holder's choice is not entirely clear. The end of Fig. 3, provides an example call european_stock_option which constructs a call option on ABC Co. In practice, functions like european_stock_option would be defined in BCL's standard library, and would be called by users who wish to model European stock options directly or who wish to model contracts that contain such options as sub-contracts.

3.1   Motivation for Gradual Types

Given that BCL is mostly used to describe financial contracts, it should come as no surprise that our users are largely financial professionals. In particular, many are financial engineers or quantitative analysts with some programming experience in dynamic languages such as Python and MATLAB. Typically these users need to translate term sheets, plain-English descriptions of a contract, into BCL for consumption by our system. These contracts are mostly one-off and, once finished, are unlikely to be reused as subcontracts to build further contracts. For these reasons, the users of BCL are primarily concerned with development speed. Ideally, they would like to be able to translate a term sheet as quickly as possible, so that they may focus on analyzing the contract's behavior once it has been ingested by our system.

On the other hand, the maintainers of BCL and its standard library are software engineers and functional programmers with extensive experience in OCaml, C++ and other static languages. The main jobs of the BCL maintainers are implementing language extensions and standard library functions. One of the significant constraints that they face is preserving backwards compatibility. All existing user contracts must continue to work as BCL evolves–even minor changes in behavior are unacceptable! Given the broad reuse of the features that BCL's language maintainers implement and the difficulties involved in rolling back features, correctness is the paramount concern of BCL maintainers.

Finally, it is important to note that the version of BCL described here is actually the second version of BCL. The first version of BCL was dynamically typed, so we will distinguish it from the second version by referring to it as Dynamic BCL. Dynamic BCL supports only a few primitive data types, as well as a list composite type; it does not support algebraic types. It also runs only minimal validation before attempting evaluation. This simplicity makes Dynamic BCL well suited to our users who seek to quickly feed contracts into our system, but ill-suited to the library code written by our maintainers. Additionally, some users who encounter runtime type errors while implementing particularly complex contracts would turn to the maintainers for assistance, further increasing the burden on the maintainers. It was in light of these issues, that we developed (Static) BCL.

To address the issues with Dynamic BCL while remaining useful to our users, BCL aims to be a static language that feels roughly dynamic. To this end, BCL supports implicit static types via type inference; we chose Hindley-Milner style inference so that our users could omit type annotations in almost all cases. BCL also supports record and variant types, although they are structural rather than the nominal ones typically seen in OCaml and Haskell. This choice also lends BCL a more dynamic feel.

The goal of BCL's design is to retain enough flexibility for our users, while introducing static types for the benefit of our language maintainers. However, "enough flexibility" is entirely subjective and some users may well feel that any amount of static checking results in a system that is too inflexible. Gradual types address this concern by allowing users to use dynamic types where they like, while also allowing maintainers to use static types where they would like. Importantly, gradual types guarantee that fully dynamic code and fully static code can co-exist, and that static code is never blamed for runtime type errors. Taken together, these two guarantees satisfy both groups, and

ensure that the type errors that dynamic users see are isolated to the code that they themselves wrote.

3.2   Core Calculus

BCL's core calculus is the lambda calculus extended with structural composite types and recursive types. Furthermore, BCL is implicitly-typed and supports Hindley-Milner style type inference. This section describes the types and terms of this core calculus. Note, however, that the grammars in this section are abstract representations of BCL's theoretical underpinnings, and do not cover the full set of productions in BCL's grammar.

$$\kappa ::= * \mid \rho \mid \kappa \Rightarrow \kappa \qquad\qquad C ::= \to \mid \Pi \mid \Sigma \mid \dots$$
$$\tau ::= \alpha \mid C \mid \tau\,\tau \mid l : \tau; \tau \mid \epsilon \mid \mu\alpha.\tau \quad \sigma ::= \tau \mid \forall\alpha.\sigma$$

Fig. 4. Grammar of types and kinds

Kinds and Types. The grammar of the types and kinds that describe BCL is given in Fig. 4. Our kind system is fairly standard and consists of only three forms. The base kind, $*$, is the kind of "proper" types–Int and $Int \to Int$, for example–which themselves describe terms. The row kind, ρ, is of course the kind for rows. The operator kind, $\Rightarrow$, is the kind of type operators – $Array$ and $\to$, for example – which take types as arguments and which do not directly describe terms.

C ranges over type constructors, including the type operators for function types ($\to$ of kind $* \Rightarrow * \Rightarrow *$), record types ($\Pi$ of kind $\rho \Rightarrow *$) and variant types (Σ of kind $\rho \Rightarrow *$). C may also include additional constructors for base types (e.g. Int and $String$) and more type operators (e.g. $Array$) as desired. However, these additional constructors are not useful for our purposes here, so we make no further mention of them.

Our type system is stratified into monomorphic types and type schemes, per [17]. Monomorphic types, τ, consist of type variables, type constructors, and record, variant and recursive types. Type variables are ranged over by α, β, γ, etc., and are explicitly bound by μ and $\forall$ types, as described below. Rows are written $l : \tau; \tau'$, indicating that the row has a field labeled l of type τ. τ' has kind ρ and dictates the other fields that the row may contain. If τ' is a type variable, the row can contain arbitrary additional fields; if τ' is the empty row, ϵ, the row contains no additional fields; finally if τ' is another type of the form $l : \tau; \tau'$, then the row contains exactly the fields specified therein. Recursive types are written $\mu\alpha.\tau$, where the variable α represents the point of recursion and is bound within τ. BCL's recursive types are equi-recursive, so it does not have explicit constructs for rolling and unrolling recursive types. Finally, type schemes have two forms: monomorphic types and universally quantified schemes. Monomorphic types, τ, are merely the types described above. Universally quantified schemes, $\forall\alpha.\sigma$, bind the variable α within the scheme σ. Naturally, it is through universal quantification that BCL supports parametric polymorphism.

$$t ::= x \mid \lambda x.t \mid t\ t \mid \textbf{let rec } x = t \textbf{ in } t \mid t : \tau \mid \{\overline{l_i : t_i}\} \mid t.l \mid l\ t \mid \textbf{match } t \textbf{ with } \overline{l_i x_i \Rightarrow t_i}$$

Fig. 5. Grammar of terms

Terms. The grammar of the terms in BCL is given in Fig. 5. Most of the term forms are drawn directly from the lambda calculus. Term variables are ranged over by x, y, z, etc., and are introduced by lambda abstraction, let-bindings and match-expressions. Lambda abstraction is written $\lambda x.t$ and binds x within the expression t. Lambda abstractions are eliminated by application, which is denoted by juxtaposition: $t\ t$. Let-bindings are written **let rec** $x = t$ **in** t. The **rec** is optional and, when present, indicates x may be referenced by the expression to the right of the $=$; x may of course always be referenced by the expression to the right of the **in**. Type annotations are written $t : \tau$, and serve to ensure that t has the they τ.

In addition to the forms described above, BCL supports records and variants. Record introduction is written $\{\overline{l_i : t_i}\}$, where t_i evaluates to the value stored in the field l_i. Records are eliminated by field projection. The projection of the field l from the record t is written $t.l$. Variant introduction is written $l\ t$, where the label l is used to tag the variant's payload, t. Variants are eliminated by case analysis, which is written **match** t **with** $\overline{l_i x_i \Rightarrow t_i}$, which evaluates to the branch specified by the tag associated with the variant t.

4   Implementation

We identify three main components required to add gradual typing to a statically typed language with type inference, such as BCL. The first is the ability to annotate terms with types, as these annotations dictate whether a term is type-checked statically or dynamically. The second is the addition of a dynamic type to the existing set of types, and the third is an algorithm to unify the existing types with the newly added dynamic type. Since our grammar, shown in Fig. 5, already supports explicit annotation of terms, we have the means to differentiate between dynamically typed and statically typed code. We add a dynamic type, ?, to our set of types; it serves to indicate that a term that will be dynamically typed. BCL's type inference algorithm statically infers a type for every term, meaning that by default BCL programs are completely statically typed. In order to tell the type system to dynamically type some terms, we must explicitly annotate those terms with the ? type.

For example: a simple increment function can be defined in BCL as follows.

```
let incr x = x + 1 in incr
```

The type system will infer the type $Int \rightarrow Int$ for the incr function. However, we can instead provide an explicit annotation.

```
let incr x = x + 1 in incr : ? -> Int
```

In this case, the inference algorithm retains the annotated type as the type of the function. Any type checks on the argument of the incr function would be put off until runtime. While the type checks pertaining to ? types are delayed, we still need to

complete the inference procedure in order to infer the types of the un-annotated portions of the program (like the return type of incr). Siek and Vacchrajani [25] (S&V) extend the standard unification-based inference algorithm to handle the ? type. Their algorithm is based on the *consistent-equal* relation which takes into consideration the type variables that are generated as part of a typical type inference algorithm. Fortunately for us, their algorithm works well for our implementation with only minor adaptations.

Figure 6 shows an outline of our adaptation of S&V's inference algorithm. Unlike the original algorithm by S&V, BCL's does not separate constraint generation and constraint solving.[2] This difference is important, as it means that our inference algorithm does not have access to the whole constraint set prior to unification. Instead the infer function traverses the term, generating and solving constraints on the fly. For example, if it encounters an application $t_1 \ t_2$, it figures out the type of the term from the environment (Γ) and generates a constraint like $\{\tau_1 \simeq \tau_2 \to \alpha\}$, where τ_1 is the type of term t_1, τ_2 is the type of t_2 and α is a fresh type variable. infer sends this constraint to unify, which attempts to satisfy it or raises an error if the constraint cannot be satisfied.

Figure 6 shows the infer case for a term t annotated with the type τ. infer generates a constraint which tries to unify the type inferred for t with the annotated type, τ. We highlight this case for two reasons. First, the only way we can currently introduce a ? type in BCL is through an annotation. Therefore, this is the only point where constraints involving the ? type originate. Second it is critically important that this case returns the annotated type and not the inferred type. Note that in incr the inferred type $Int \to Int$ differs from–but is *consistent-equal* with–the annotated type $? \to Int$. We always want the user's explicit annotation to take precedence in this situation.

BCL's unification algorithm is already based on Huet's unification algorithm, which makes adopting the changes suggested by S&V easier. The crux of S&V's algorithm lies in the way the ? type unifies with other types, and particularly with type variables. When ? unifies with a type variable, S&V's algorithm makes ? the representative node. However, when ? unifies with types other than type variables, the other type becomes the representative element of the resulting set. The find and merge functions in Fig. 6 come from the union-find data structure that underlies Huet's unification algorithm. Respectively, they compute a node's representative element, and union two nodes' sets keeping the representative element of the first node's set.

The first six cases of the unify function handle unification with the ? type as laid out by S&V. We say first six because Cases 1 and 2 take care of unifying the ? type with type variables as specified by S&V's algorithm. Cases 3 and 4 handle an edge case in their algorithm. These two cases simulate the operational semantics of Siek and Taha [23], which require constraints like $\{? \simeq \alpha \to \beta\}$ to be treated as $\{? \to ? \simeq \alpha \to \beta\}$. We use new to create a new node different from what was passed in to handle this case.

Cases 8–11 take care of unifying with row and recursive types, neither of which are covered by S&V's solution. However, it is our observation that these types do not require special handling. A constraint like $\{x \ : \ Int; \epsilon \ \simeq \ ?\}$ would be handled by

---

[2] Put another way, our inference algorithm solves each constraint immediately after generating it, and before generating the next constraint.

```
maybe_copy_dyns (τ₁ ≃ τ₂) =
  τ₁′ ← if was_copied τ₁ then τ₁ else copy_dyn τ₁
  τ₂′ ← if was_copied τ₂ then τ₂ else copy_dyn τ₂
  τ₁′ ≃ τ₂′

unify τ₁″ τ₂″ =
  τ₁ ← find τ₁″
  τ₂ ← find τ₂″
  if was_visited τ₁ and was_visited τ₂ then
    ()
  else case maybe_copy_dyns (τ₁ ≃ τ₂) of
      α ≃ τ  |  τ ≃ α  ⇒ merge τ α  (∗ Case 1 & 2 ∗)
    |  ? ≃ τ₁ → τ₂  |  τ₁ → τ₂ ≃ ?  ⇒  (∗ Case 3 & 4 ∗)
      unify τ₁ (new ?)
      unify τ₂ (new ?)
    |  ? ≃ τ  |  τ ≃ ?  ⇒ merge τ ?  (∗ Case 5 & 6 ∗)
    |  τ₁₁ → τ₁₂ ≃ τ₂₁ → τ₂₂  ⇒  (∗ Case 7 ∗)
      unify τ₁₁ τ₂₁
      unify τ₁₂ τ₂₂
    |  l : τ₁; τ₂ ≃ l′ : τ₁′; τ₂′  if  l = l′  ⇒  (∗ Case 8 ∗)
      unify τ₁ τ₁′
      unify τ₂ τ₂′
    |  l : τ₁; τ₂ ≃ l′ : τ₁′; τ₂′  ⇒  (∗ Case 9 ∗)
      α ← fresh_type_variable()
      unify (l : τ₁; α) τ₂′
      unify (l′ : τ₁′; α) τ₂
    |  μα.τ ≃ τ′  |  τ′ ≃ μα.τ  ⇒  (∗ Case 10 & 11 ∗)
      mark_visited (μα.τ)
      unify τ[μα.τ/α] τ′
    |  ε ≃ ε  ⇒  ()  (∗ Case 12 ∗)
    |  _  ⇒ error

infer Γ t =
  case t of
    ...
    t : τ →
      case (unify (infer Γ t) τ) of
        Error ⇒ Error : inconsistent types
        |  _ ⇒ τ
```

Fig. 6. Type inference algorithm

Case 2 and ? would be merged with the row type $x : Int; \epsilon$. Now suppose the ? is present inside the row type like in the following constraint $\{x : ?; \epsilon \simeq x : Int; \epsilon\}$; this will be handled by Case 8 and then Cases 5 and 12 when we recursively call unify with the types within the row. The same holds true for unification with the recursive type. For example, a constraint like $\{List\ Int \simeq List\ ?\}$ will have the following successful unification trace:

$$\{List\ Int \simeq List\ ?\} \tag{Case 10}$$
$$\rightarrow \{\Pi(head : Int; tail : List\ Int; \epsilon) \simeq List\ ?\} \tag{Case 11}$$
$$\rightarrow \{\Pi(head : Int; tail : List\ Int; \epsilon) \simeq \Pi(head : ?; tail : List\ ?; \epsilon)\} \tag{Case 8}$$
$$\rightarrow \{\{Int \simeq ?\}, \{List\ Int \simeq List\ ?\}\} \tag{Case 6, Case 10}$$
$$\rightarrow \cdots$$
$$\rightarrow \{\Pi\ \epsilon \simeq \Pi\ \epsilon\} \tag{Case 8}$$
$$\rightarrow \{\epsilon \simeq \epsilon\} \rightarrow () \tag{Case 12}$$

Where $List$ is defined as follows.

$List\ \alpha \equiv \mu a.\Sigma(Nil : \Pi\epsilon; Cons : \Pi(head : \alpha; tail : a; \epsilon); \epsilon)$ Note that BCL supports *equi-recursive types*, as mentioned in Sect. 3, so `unify` tracks the types it visits with `mark_visited` and `was_visited` to detect cycles.

The `copy_dyn` Conundrum: The `copy_dyn` function is a crucial part of the way ? unifies with other types. In S&V's presentation, `copy_dyn` ensures that each ? node in the constraint set is physically unique. Without this step, multiple types might unify with the same ? node, and then transitively with each other. This has the potential to cause spurious failures in `unify`. S&V's solution to this is to traverse the constraint set and duplicate each ? node prior to unification; this is performed by their implementation of `copy_dyn`. Unfortunately, we do not have access to the full constraint set, because our inference algorithm generates and solves constraints in one step.

Our first attempt at working around this issue was to call `copy_dyn` as the first step in unification. However, this leads to over copying. For example, consider the constraint $\{ ? \rightarrow \alpha \simeq \alpha \rightarrow \tau \}$. According to Case 7 of `unify`, when α unifies with the ? node, `copy_dyn` is called and a new ? node is created in the union-find structure. But when α then unifies with τ, `find` looks up ? as α's representative element, and `copy_dyn` is called once more. τ therefore unifies with the new ? node, instead of the one which unified with α. Thus, we lose the fact that τ and α are the same type.

To rectify this, we implement `maybe_copy_dyns`, which traverses a constraint and copies each ? node exactly once.[3] The result of this is the same as originally intended by S&V's `copy_dyn` function. That is, we ensure there is a unique ? node in the union-find structure for every unique use of ?.

4.1   Discussion

In Sect. 2 we gave an overview of how statically typed languages approach this problem of promoting dynamic typing. It is our observation that most statically typed, object-oriented languages approach dynamic typing following Abadi et al. That is, their dynamic type exploits subtype polymorphism to bypass static type checking. This is a natural direction for object-oriented languages which rely heavily on subtyping. In order to inspect the types at runtime, these languages make use of type reflection. Java is one such language where work has been done to add dynamic types using reflection,

---

[3] There are many ways to accomplish this. Our approach was to use one canonical ? node in type annotations, and compare each ?'s address to the canonical node's address before copying.

contracts and mirrors [6]. The Java Virtual Machine supports many dynamic languages like Jython and JRuby, demonstrating that such runtime constructs help static languages add more dynamic type checks. However, these implementations only add dynamic checks, and do not achieve the gradual goal of a seamless mix of static and dynamic types as in [26]. To our knowledge, only Featherweight Java [11] has attempted to support proper gradual typing [12]. In any case, the primary purpose for dynamic types in these languages is inter-operation with other dynamic languages. This differs from our own purpose and the end result does not fit our needs well. Thus we conclude that this approach was not a good design choice for us.

The languages closest to BCL are statically typed functional languages with type inference, such as SML, OCaml, and Haskell. OCaml has incorporated dynamic typing at the library level by leveraging its support for generalized algebraic data types [3]. Similarly, Haskell supports a dynamic type as a derivative of the `Typeable` type class, which uses reflection [13] to look at the runtime representation of types. While these approaches introduce more dynamism, they lack the simplicity of gradual typing, which hide all the nuts and bolts of the type system under a simple ? annotation.

Seamless interoperation of static and dynamic types as promised by gradual typing fits very well with our use case. It lets our end users access both paradigms without knowledge of specialized types or constructs. Furthermore, the approach we use— extending unification-based inference with gradual typing—is a natural extension for languages like BCL, which support static type inference. The addition of dynamic types to the type system easily boils down to how we handle this new type in the unification algorithm, and does not require reworking the entire type system. We attribute this benefit to S&V's proposed inference algorithm, which incorporates the essence of the $\lambda_{\rightarrow}^{?}$ type system. This makes it easier to adapt to an existing language with similar constructs.

Garcia and Cimini's work takes a different approach to this problem but their end goal is the same: gradual type inference in a statically typed language. The authors of that work feel that S&V's approach has "complexities that make it unclear how to adopt, adapt, and extend this approach with modern features of implicitly typed languages like let-polymorphism, row-polymorphism and first class polymorphism". Our experience with S&V's approach was different: we found the integration fairly simple without major changes to the original inference algorithm. We leave a deep dive into the differences between these two schemes to future work. Based on Garcia and Cimini's design principle, Xie et al. [34] introduce an inference algorithm with support for higher-rank polymorphism, using a *consistent subtyping* relation. In contrast, BCL only infers rank-1 polymorphic types and doesn't support higher-rank polymorphism.

We recognize an added benefit of going from a static to a dynamic language with explicit ? annotations. Promoting static type checking in a dynamic language without type inference requires the programmer to add annotations to all parts of the code that they want statically checked. Needing to add these annotations is such a burden for the programmer that they often skip some annotations and miss out on static optimizations. These un-annotated types are implicitly dynamic, leading to runtime overhead, despite the fact that on many occasions they could be statically inferred. This in turn has lead to efforts to making gradual typing more efficient [22].

BCL does not have this issue as it provides static inference by default. It therefore enjoys the optimizations of static typing and can skip unnecessary runtime checks.

Moreover, BCL could support a dynamic-by-default mode with an additional compiler flag that implicitly annotates un-annotated terms with the `?` type. This makes it even more seamless to go from complete static typing to complete dynamic typing. We might also consider doing this implicit annotation on a file-level or function-level. In cases where it is possible to separate dynamic and static components, this could even lead to cleaner refactoring. These ideas have not yet been implemented in BCL but are something we intend to do as future work.

5   Application of Gradual Types

Gradual typing enables the quick prototyping common in dynamic languages, as well as specialized applications that enable simplification of existing code. In this section, we focus on the latter, due to space constraints. Notice, in Fig. 3, that the `scale` combinator [19] is simply multiplication of a contract by a floating-point *observable*. In a domain specific language like BCL, it is convenient to reuse the `**` multiplication syntax for `scale` as well. We can fit this more general *observable* multiplication operator into the type system with the gradual type $Obs\ Double \rightarrow\ ? \rightarrow\ ?$. Our new multiplication operator can delegate to `scale` when the second argument a $Contract$ at runtime and continue with *observable* multiplication, or raise a runtime type error based on the runtime type of the second argument. With this new operator, the `receive` function can be rewritten thus:

```
let receive currency amount = one currency ** amount in ...
```

There are a variety of extensions to Hindley-Milner that enable this sort of ad-hoc polymorphism statically. Type classes, for example, extend the signature of overloaded functions with classes [31], which our users would need to learn. Similarly, modular implicits introduce a separate syntax for implicit modular arguments [33]. However, these constructs require effort to educate our end users in their use and detract from the dynamic feel of the language. Gradual types, by contrast, are much easier for our end users since they already work in a dynamic environment and it does not require new syntax (save a `?` annotation).

It is worth noting that, while the new multiplication operator can be given a valid type in BCL, it cannot currently be implemented in BCL; it can only be implemented as a built-in operator because BCL provides no way to perform case analysis on the runtime type of a dynamic value. However, addressing this is actually quite easy if we reuse BCL's existing support for variants. That is, we could implement a `dynamic_to_type` primitive which consumes a dynamic value and produces a variant describing its runtime type. This would allow us to then branch on this variant with the existing **match** construct. Figure 7 shows a prototype of a function that achieves this effect assuming the `dynamic_to_type` primitive is defined.

`dynamic_to_type` is interesting in light of our discussion in Sect. 2, which describes dynamic programming as the territory of BCL's users and not its maintainers. Clearly, however, the dynamic multiplication operator is something that would live in BCL's standard library and be maintained by the language maintainers. Indeed there are a number of interesting standard library functions which we might add on top of

```
let dyn_obs_mul x y = match dynamic_to_type (y) with
  | Obs Double => x ** y
  | Contract => scale x y
in
    dyn_obs_mul : Obs Double → Dyn → Dyn
```

Fig. 7. Sample of a dynamic observable multiplication function

dynamic_to_type. Another simple example would be a any_to_string function, which could produce a string representation for arbitrary types by traversing their runtime type and delegating to the appropriate type-specific to-string function. Such a function would be very handy for debugging and quick inspection of values.

The any_to_string example is a function which consumes an arbitrary value. However, there are equally compelling use cases for producing arbitrary values. For example, property-based testing frameworks rely on automatically generating values that conform to certain constraints. We could implement a simple property-based testing framework with a function which consumes the output of dynamic_to_type and generates arbitrary values that conform to that type. Such a framework would be especially useful in a domain such as ours, where real money is at stake, and where robust testing is absolutely critical.

6   Conclusion

Dynamic languages are extremely popular with many users. For users with a limited computer science background, for whom ease-of-use is the paramount, this is doubly true. However, despite the flexibility offered by dynamic typing, the safety offered by static typing is helpful in domains where correctness is critical. In such an arena, gradual types are a perfect blend of both paradigms, and they provides a middle ground to please a larger group of users. Given this, it is important for the literature to speak about adapting gradual types to existing languages. As a first step towards that, we write about our experiences adapting gradual typing to our implementation of a statically typed functional language with type inference. We provide context in terms of how others in similar situations approached this problem, and we elaborate our inference algorithm with key insights around what worked for us and what did not. We identify an interesting use case for gradual types here at Bloomberg, where we look to harmonize end users and language maintainers with competing goals. End users want to specify financial contracts without worrying about static typing demands, while language maintainers need a more rigorous type system that ensures that libraries that they write are error-free. Gradual types allow us to satisfy both groups. We also intend to gather feedback from our end users and maintainers about how gradual types are being used, which can give insight into possible tweaks to make this system more amenable to all.

References

1. Using Type Dynamic. https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/types/using-type-dynamic. Accessed 02 Jan 2021
2. Abadi, M., Cardelli, L., Pierce, B., Plotkin, G.: Dynamic typing in a statically typed language. ACM Trans. Program. Lang. Syst. (TOPLAS) **13**(2), 237–268 (1991)
3. Balestrieri, F., Mauny, M.: Generic programming in OCaml. Electron. Proc. Theor. Comput. Sci. **285**, 59–100 (2018). https://doi.org/10.4204/EPTCS.285.3
4. Damas, L., Milner, R.: Principal type-schemes for functional programs. In: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 207–212 (1982)
5. Garcia, R., Cimini, M.: Principal type schemes for gradual programs. SIGPLAN Not. **50**(1), 303–315 (2015). https://doi.org/10.1145/2775051.2676992
6. Gray, K.E., Findler, R.B., Flatt, M.: Fine-grained interoperability through mirrors and contracts. ACM SIGPLAN Not. **40**(10), 231–245 (2005)
7. Greenberg, M.: The dynamic practice and static theory of gradual typing. In: 3rd Summit on Advances in Programming Languages (SNAPL 2019). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik (2019)
8. Gronski, J., Knowles, K., Tomb, A., Freund, S.N., Flanagan, C.: Sage: hybrid checking for flexible specifications. In: Scheme and Functional Programming Workshop, vol. 6, pp. 93–104 (2006)
9. Hejlsberg, A.: C# 4.0 and beyond by anders hejlsberg. Microsoft Channel, vol. 9 (2010)
10. Huet, G.: Resolution d'equations dans des languages d'order 1, 2,..., ω. these d'etat, University de Paris (1976). https://ci.nii.ac.jp/naid/10006536039/en/
11. Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst. **23**(3), 396–450 (2001). https://doi.org/10.1145/503502.503505
12. Ina, L., Igarashi, A.: Gradual typing for featherweight Java. Comput. Softw. **26**(2), 18–40 (2009)
13. Peyton Jones, S., Weirich, S., Eisenberg, R.A., Vytiniotis, D.: A reflection on types. In: Lindley, S., McBride, C., Trinder, P., Sannella, D. (eds.) A List of Successes That Can Change the World. LNCS, vol. 9600, pp. 292–317. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30936-1_16
14. Knight, K.: Unification: a multidisciplinary survey. ACM Comput. Surv. (CSUR) **21**(1), 93–124 (1989)
15. Matthews, J., Findler, R.B.: Operational semantics for multi-language programs. ACM Trans. Program. Lang. Syst. (TOPLAS) **31**(3), 1–44 (2009)
16. Meijer, E., Drayton, P.: Static typing where possible, dynamic typing when needed: the end of the cold war between programming languages. Citeseer (2004)
17. Milner, R.: A theory of type polymorphism in programming. J. Comput. Syst. Sci. **17**(3), 348–375 (1978)
18. Miyazaki, Y., Sekiyama, T., Igarashi, A.: Dynamic type inference for gradual Hindley-Milner typing. Proc. ACM Program. Lang. **3**(POPL), 1–29 (2019). https://doi.org/10.1145/3290331
19. Peyton Jones, S., Eber, J.M., Seward, J.: Composing contracts: an adventure in financial engineering (functional pearl). SIGPLAN Not. **35**(9), 280–292 (2000). https://doi.org/10.1145/357766.351267
20. Pottier, F.: A modern eye on ml type inference: old techniques and recent developments (September 2005)
21. Pottier, F., Rémy, D.: The essence of ML type inference, pp. 389–489 (January 2005)

22. Rastogi, A., Chaudhuri, A., Hosmer, B.: The ins and outs of gradual type inference. SIG-PLAN Not. **47**(1), 481–494 (2012). https://doi.org/10.1145/2103621.2103714
23. Siek, J., Taha, W.: Gradual typing for functional languages. In: Scheme and Functional Programming Workshop (2006)
24. Siek, J., Taha, W.: Gradual Typing for objects. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 2–27. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73589-2_2
25. Siek, J.G., Vachharajani, M.: Gradual typing with unification-based inference. In: Proceedings of the 2008 Symposium on Dynamic Languages, pp. 1–12 (2008)
26. Siek, J.G., Vitousek, M.M., Cimini, M., Boyland, J.T.: Refined criteria for gradual typing. In: Ball, T., Bodik, R., Krishnamurthi, S., Lerner, B.S., Morrisett, G. (eds.) 1st Summit on Advances in Programming Languages (SNAPL 2015). Leibniz International Proceedings in Informatics (LIPIcs), vol. 32, pp. 274–293. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, Dagstuhl (2015). https://doi.org/10.4230/LIPIcs.SNAPL.2015.274
27. Tarjan, R.E.: Efficiency of a good but not linear set union algorithm. J. ACM (JACM) **22**(2), 215–225 (1975)
28. Tobin-Hochstadt, S., Felleisen, M.: Interlanguage migration: from scripts to programs. In: Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications, pp. 964–974 (2006)
29. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of typed scheme. ACM SIGPLAN Not. **43**(1), 395–406 (2008)
30. Vitousek, M.M., Kent, A.M., Siek, J.G., Baker, J.: Design and evaluation of gradual typing for Python. In: Proceedings of the 10th ACM Symposium on Dynamic Languages, DLS 2014, pp. 45–56. Association for Computing Machinery, New York (2014). https://doi.org/10.1145/2661088.2661101
31. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1989, pp. 60–76. Association for Computing Machinery, New York (1989). https://doi.org/10.1145/75277.75283
32. Wand, M.: A simple algorithm and proof for type inference (1987)
33. White, L., Bour, F., Yallop, J.: Modular implicits. Electron. Proc. Theor. Comput. Sci. **198**, 22–63 (2015). https://doi.org/10.4204/EPTCS.198.2
34. Xie, N., Bi, X., Oliveira, B.C.D.S., Schrijvers, T.: Consistent subtyping for all. ACM Trans. Program. Lang. Syst. **42**(1), 1–79 (2019). https://doi.org/10.1145/3310339