¡s¿ E fficient Grad ual Typ ing Andre Kuh l ens ch midt De ya ae ld een Alm ah all awi Jeremy G . Sie k Indiana University ak uh l ens @ ind iana . edu Indiana University d alm ah al @ ind iana . edu Indiana University j sie k @ ind iana . edu ar X iv : 18 02 . 06 375 v 1 [cs . PL] 18 Feb 2018 Abstract To date , implementations of gradual typing have only delivered two of these three properties . For example , Typ ed R acket [48] provides sound ness and interoper ability but suffers from slow downs of up to 100 [45 , 46] on a partially typed program . Thorn [10 , 50] and Safe Type Script [34] provide better performance but limit interoper ability . Type Script [9 , 27] and Grad ual talk [2 4] do not provide sound ness and their performance is on par with dynamic languages but not static ones , but they provide seamless interoper ability . Several papers at O OP SL A 2017 begin to address the efficiency concerns for gradually typed languages that are committed to sound ness and interoper ability . Ba uman et al . [7] demonstrate that a tracing J IT can eliminate 90 % of the over heads in Typ ed R acket due to gradual typing . Richards et al . [35] augment the H iggs J IT compiler and virtual machine (VM) [13] for JavaScript , re - pur posing the VM s notion of shape to implement mon ot onic references [42]. Richards et al . [35] reports that this reduces the worst slow downs to 45 %, with an average slowdown of just 7 %. Meanwhile , M ue hl bo e ck and Tate [33] show that for nom inally - ty ped object - oriented languages , efficiency is less of a problem . In this paper we demonstrate that efficient gradual typing can be achieved in struct urally - ty ped languages by relatively straightforward means . We build and evaluate an ahead - oft ime compiler that uses carefully chosen runtime representations and implements two important ideas from the theory of gradual typing . It uses space efficient coerc ions [20 , 30 , 40 , 43] to implement casts and it reduces overhead in statically typed code by using mon ot onic references [42]. Grad ual typing combines static and dynamic typing in the same program . One would hope that the performance in a gradually typed language would range between that of a dynamically typed language and a statically typed language . Ex isting implementations of gradually typed languages have not achieved this goal due to over heads associated with runtime casts . Tak ikawa et al . (2016) report up to 100 slow downs for partially typed programs . In this paper we present a compiler , named Gr ift , for evaluating implementation techniques for gradual typing . We take a straightforward but surprisingly unexpl ored implementation approach for gradual typing , that is , ahead - of - time compilation to native assembly code with carefully chosen runtime representations and space - efficient coerc ions . Our experiments show that this approach achieves performance on par with O Cam l on statically typed programs and performance between that of Gam bit and R acket on unt yp ed programs . On partially typed code , the geometric mean ranges from 0 . 42 to 2 . 36 that of (un ty ped) R acket across the benchmarks . We implement casts using the coerc ions of Sie k , Th iem ann , and Wad ler (2015). This technique eliminates all catastrophic slow downs without introducing significant overhead . Across the benchmarks , coerc ions range from 15 % slower (ff t) to almost 2 faster (mat mult) than regular casts . We also implement the mon ot onic references of Sie k et al . (2015). Mon ot onic references eliminate all overhead in statically typed code , and for partially typed code , they are faster than prox ied references , sometimes up to 1 . 48 . Contributions This paper makes these contributions . 1 Introduction A space - efficient semantics for mon ot onic references and lazy - D coerc ions (Section 3). The first ahead - of - time compiler , named Gr ift , for a gradually typed language that targets native assembly code . The compiler is the first to implement space efficient coerc ions (Section 4). Exper iments (Section 5 . 2) showing performance on statically typed code that is on par with O Cam l , performance on dynamically typed code that is between Gam bit and R acket , and performance on partially typed code ranging from 0 . 42 to 2 . 36 that of R acket . Exper iments showing that coerc ions eliminate catastrophic slow downs without adding significant overhead (Section 5 . 3). Grad ual typing combines static and dynamic type checking in the same program , giving the programmer control over which typing discipline is used for each region of code [5 , 24 , 32 , 39 , 47]. We would like gradually typed languages to be efficient , sound , and provide interoper ability . Regarding efficiency , we would like the performance of gradual typing to range from being similar to that of a dynamically typed language to that of a statically typed language . Regarding sound ness , programmers (and comp ilers) would like to trust type annotations and know that runtime values respect their compile - time types . Third , regarding interoper ability , static and dynamic regions of code should interoper ate seamlessly . PL 17 , January 01 03 , 2017 , New York , NY , USA 2017 . https :// doi . org / 0000 001 . 0000 001 1 PL 17 , January 01 03 , 2017 , New York , NY , USA Andre Kuh l ens ch midt , De ya ae ld een Alm ah all awi , and Jeremy G . Sie k Exper iments showing that mon ot onic references eliminate overhead in statically typed code (Section 5 . 4). Param eter m of mod inv has type Dyn , but b of eg cd has type Int , so there is an implicit cast from Dyn to Int . With gradual typing , this implicit cast comes with a runtime cast that will trigger an error if the input to this program is a string . This runtime cast is required to ensure sound ness : without it a string could flow into eg cd and mas querade as an Int . Sound ness is not only important for software engineering reasons but it also impacts efficiency both positively and negatively . Ens uring sound ness in the presence of first - class functions and mut able references is nont riv ial . When a function is cast from Dyn to a type such as Int Int , it is not possible for the cast to know whether the function will return an integer on all inputs . Instead , the standard approach is to wrap the function in a proxy that checks the return value each time the function is called [18]. Similarly , when a mut able reference is cast , e . g ., from Ref Int to Ref Dyn , the reference is wrapped in a proxy that casts from Int to Dyn on every read and from Dyn to Int on every write [29 , 30]. Section 2 provides background on gradual typing , focusing on runtime casts and the tension between efficiency , sound ness , and interoper ability . 2 T ensions in Grad ual Typ ing From a language design perspective , gradual typing touches both the type system and the operational semantics . The key to the type system is the consistency relation on types , which enables implicit casts to and from the unknown type , here written Dyn , while still catching static type errors [5 , 24 , 39]. The dynamic semantics for gradual typing is based on the semantics of contracts [18 , 21], coerc ions [28], and inter language migration [32 , 47]. Because of the shared mechanisms with these other lines of research , much of the ongoing research in those areas benefits the theory of gradual typing , and vice versa [14 16 , 22 , 23 , 25 , 31 , 44]. In the following we give a brief introduction to gradual typing by way of an example that emphasizes the three main goals of gradual typing : supporting interoper ability , sound ness , and efficiency . Efficiency Ideally , statically typed code within a gradually typed program should execute without overhead . Likewise , partially typed or unt yp ed code should execute with no more overhead than is typical of dynamically typed languages . Consider the eg cd on the right side of Figure 1 . Inside this eg cd , the expression (mod ulo b a) should simply compile to an id iv instruction (on x 86). However , if the language did not ensure sound ness as discussed above , then this efficient compilation strategy would result in undefined behavior (se gment ation faults at best , hacked systems at worst). It is sound ness that enables type - based specialization . However , sound ness comes at the cost of the runtime casts at the boundaries of static and dynamic code . Inter oper ability and Evolution Consider the example program in Figure 1 , written in a variant of Typ ed R acket that we have extended to support fine - gr ained gradual typing . On the left side of the figure we have an unt yp ed function for the extended greatest common div is or . With gradual typing , un annot ated parameters are dynamically typed and therefore assigned the type Dyn . On the right side of the figure is the same function at a later point in time in which the parameter types have been specified (Int) but not the return type . With gradual typing , both programs are well typed because implicit casts are allowed to and from Dyn . For example , on the left we have the expression (mod ulo b a), so b and a are implicitly cast from Dyn to Int . On the right , there is an implicit cast around (list b 0 1) from (List Int) to Dyn . The reason that gradual typing allows implicit casts both to and from Dyn is to enable evolution . As a programmer adds or removes type annotations , the program continues to type check and also exhibits the same behavior up to cast errors , a property called the gradual guarantee [41]. 3 Sem antics of a Grad ual Language The type system of Gr ift s input language is the standard one for the gradually typed lambda calculus [30 , 36 , 39]. The operational semantics , as usual , is expressed by a translation to an ¡/s¿