

52 Efficient Gradual Typing Andre Kuhlenchmidt, Debra Allen, Jeremy G. Siek Indiana University al@indiana.edu, al@indiana.edu, j.siek@indiana.edu arXiv:1802.06375v1 [cs.PL] 18 Feb 2018 Abstract To date, implementations of gradual typing have only delivered two of these three properties. For example, Typed Racket [48] provides soundness and interoperability 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 interoperability. Type Script [9, 27] and Gradualtalk [2–4] do not provide soundness and their performance is on par with dynamic languages but not static ones, but they provide seamless interoperability. Several papers at OOPSLA 2017 begin to address the efficiency concerns for gradually typed languages that are committed to soundness and interoperability. Bauman et al. [7] demonstrate that a tracing JIT can eliminate 90% of the overheads in Typed Racket due to gradual typing. Richards et al. [35] augment the Higgs JIT compiler and virtual machine (VM) [13] for JavaScript, re-purposing the VM’s notion of shape to implement monotonic references [42]. Richards et al. [35] reports that this reduces the worst slow downs to 45%, with an average slowdown of just 7%. Meanwhile, Muehlboeck and Tate [33] show that for nominally-typed object-oriented languages, efficiency is less of a problem. In this paper we demonstrate that efficient gradual typing can be achieved in structurally-typed languages by relatively straightforward means. We build and evaluate an ahead-of-time compiler that uses carefully chosen runtime representations and implements two important ideas from the theory of gradual typing. It uses space efficient coercions [20, 30, 40, 43] to implement casts and it reduces overhead in statically typed code by using monotonic references [42]. Gradual 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. Existing implementations of gradually typed languages have not achieved this goal due to overheads associated with runtime casts. Takikawa et al. (2016) report up to 100 slow downs for partially typed programs. In this paper we present a compiler, named Grift, for evaluating implementation techniques for gradual typing. We take a straightforward but surprisingly unexplored implementation approach for gradual typing, that is, ahead-of-time compilation to native assembly code with carefully chosen runtime representations and space-efficient coercions. Our experiments show that this approach achieves performance on par with OCaml on statically typed programs and performance between that of Gambit and Racket on untyped programs. On partially typed code, the geometric mean ranges from 0.42 to 2.36 that of (untyped) Racket across the benchmarks. We implement casts using the coercions of Siek, Thiemann, and Wadler (2015). This technique eliminates all catastrophic slow downs without introducing significant overhead. Across the benchmarks, coercions range from 15% slower (fft) to almost 2x faster (matmult) than regular casts. We also implement the monotonic references of Siek et al. (2015). Monotonic references eliminate all overhead in statically typed code, and for partially typed code, they are faster than proxied references, sometimes up to 1.48x. Contributions This paper makes these contributions. 1 Introduction A space-efficient semantics for monotonic references and lazy-D coercions (Section 3). The first ahead-of-time compiler, named Grift, for a gradually typed language that targets native assembly code. The compiler is the first to implement space efficient coercions (Section 4). Experiments (Section 5.2) showing performance on statically typed code that is on par with OCaml, performance on dynamically typed code that is between Gambit and Racket, and performance on partially typed code ranging from 0.42 to 2.36 that of Racket. Experiments showing that coercions eliminate catastrophic slow downs without adding significant overhead (Section 5.3). Gradual 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 interoperability. 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 soundness, programmers (and compilers) would like to trust type annotations and know that runtime values respect their compile-time types. Third, regarding interoperability, static and dynamic regions of code should interoperate seamlessly. PL 17, January 01–03, 2017, New York, NY, USA 2017. https://doi.org/0000001.0000001.1 PL 17, January 01–03, 2017, New York, NY, USA Andre Kuhlenchmidt, Debra Allen, and Jeremy G. Siek Experiments showing that monotonic references eliminate overhead in statically typed code (Section 5.4). Parameter m of modinv has type Dyn, but b of egcd 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 soundness: without it a string could flow into egcd and masquerade as an Int. Soundness is not only important for software engineering reasons but it also impacts efficiency both positively and negatively. Ensuring soundness in the presence of first-class functions and mutable references is nontrivial. When a function is cast from Dyn to a type such as Int or 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 mutable 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, soundness, and interoperability. 2 Tensions in Gradual Typing 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], coercions [28], and interlanguage 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 interoperability, soundness, and efficiency. Efficiency Ideally, statically typed code within a gradually typed program should execute without overhead. Likewise, partially typed or untyped code should execute with no more overhead than is typical of dynamically typed languages. Consider the egcd on the right side of Figure 1. Inside this egcd, the expression (modulo b a) should simply compile to an idiv instruction (on x86). However, if the language did not ensure soundness as discussed above, then this efficient compilation strategy would result in undefined behavior (segmentation faults at best, hacked systems at worst). It is soundness that enables type-based specialization. However, soundness comes at the cost of the runtime casts at the boundaries of static and dynamic code. Interoperability and Evolution Consider the example program in Figure 1, written in a variant of Typed Racket that we have extended to support fine-grained gradual typing. On the left side of the figure we have an untyped function for the extended greatest common divisor. With gradual typing, unannotated 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 (modulo 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 Semantics of a Gradual Language The type system of Grift’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 i/sj