A space-efficient call-by-value virtual machine for gradual set-theoretic types

Giuseppe Castagna
CNRS

Guillaume Duboc
ENS Lyon

Victor Lanvin
Université de Paris

Jeremy G. Siek
Indiana University

ACM Reference Format:
Giuseppe Castagna, Guillaume Duboc, Victor Lanvin, and Jeremy G. Siek. 2020. A space-efficient call-by-value virtual machine for gradual set-theoretic types. In *Proceedings of International Symposium on Implementation and Application of Functional Languages (IFL'19).* ACM, Singapore, Singapore, USA, 12 pages. https://doi.org/10.1145/3412932.3412940

Abstract. We describe the design and implementation of a virtual machine for programming languages that use gradual typing with set-theoretic types focusing on practical issues such as runtime space efficiency, and efficient implementation of tail recursive calls.

1 INTRODUCTION

Gradual typing is an approach proposed by Siek and Taha [23] to combine the safety guarantees of static typing with the programming flexibility of dynamic typing. In gradually typed programs some parts of a program may be given types and their correctness is checked at compile time (which is static typing), while some other parts are left untyped and any eventual type errors are reported at run-time (which is dynamic typing). Programmers may specify which parts are which by using suitable type annotations. A gradually-typed program is then rejected at compile time only if the statically typed parts do not type-check and/or if there exists some dynamically typed parts for which there are always failures at runtime (e.g., the application of a number to an argument).

Recently, gradual typing has received a lot of attention both from academia and industry. It is becoming the standard approach for adding static typing to dynamic languages such as, among others, JavaScript [2, 9, 20, 21], PHP [10], Python [19, 30], Clojure [3], and Scheme [28, 29]. These designs often include the use of union, intersection and, to a lesser extent, negation types (the *set-theoretic types* of the title), because such types are needed to capture many programming patterns common in dynamic languages. For instance, union and intersection types are present (in more or less limited forms) and heavily used in TypeScript [2], Flow [9] (two gradually-typed versions of JavaScript by Microsoft and Facebook, respectively) and Typed Racket [29] (a gradually-typed version of Scheme), while support for negation types are in a pull request for TypeScript. The compilers of these latter extensions do not implement gradual typing to its full extent. As we explain later, for a gradually typed language to be "sound", the compiler must insert dynamic type checks (called *type casts*) in the compiled code at the boundaries between the untyped and typed parts of the program. These checks provide strong type-based guarantees that can be used by the programmer and the compiler to reason about and optimize statically typed code. Instead, TypeScript, Flow, and Typed Racket forgo these runtime checks, and compile both the typed and untyped parts to an underlying untyped language, which, as usual, performs pervasive runtime checking. While this approach still guarantees that programs "do not get stuck", it does not provide the strong type-based guarantees of sound gradual typing. For example, it allows a number to mascarade as a string inside of statically typed code!

There are two main reasons for not implementing gradual typing thoroughly. One is somehow philosophical: you may not want to change the semantics of the dynamically typed language on which gradual types are grafted (especially when this is a very popular language such as JavaScript); type annotations are then just debugging and documentation glosses that must not alter the program they are added to. The other is eminently practical: the addition of dynamic type checks in the run-time code may hugely penalize performance [27]; to limit this impact special care must be taken in defining the compilation of untyped code and the execution of the compiled code [1, 18].

This degradation of performances for sound gradual typing has a main culprit: the use of higher order functions in untyped code which yields a compiled code where dynamic type tests are applied to unknown functions. As we explain in Section 2.3, the only reasonable way to cast a function to a particular type is to delay the runtime check to the moment of its application and check that its argument and result are both of the expected types. This delayed checking is accomplished by proxying the function, and when a function passes through many casts, it would naively be wrapped in a long chain of proxies, making performance utterly degrade. Resolving this problem requires specific implementation techniques [18].

The degradation of performances just described is dramatically amplified by the presence of union and intersection types: while in their absence determining the type expected for the input and the output of a function is just the matter of reading a couple of fields, the presence of union and intersection type makes this operation more complicated and computationally demanding; and since this computation must be performed at runtime, it adds a further critical overhead to the performance bottleneck of sound gradual typing. No implementation techniques have been developed to handle this further overhead, yet.

To summarize, gradual typing is the approach chosen by major IT actors to inject a dose of static typing into dynamic languages, but this injection also requires the use of union and intersection types. To provide stronger soundness guarantees the compiler must insert dynamic checks into the compiled code, but this may dramatically

IFL '19, September 25–27, 2019, Singapore, Singapore
© 2019 Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7562-7/19/09…$15.00
https://doi.org/10.1145/3412932.3412940

degrade performances. While compilation and implementation techniques exist for standard type structures, there is no equivalent for union and intersection types. This lack constitutes an objective obstacle to the adoption of sound gradual typing for dynamic languages. This work aims at filling this gap, by defining a virtual machine that copes with the overhead of using set-theoretic types. Before presenting the formal development we discuss some examples to give a more detailed account of the context.

Gradual Typing. The main idea of gradual typing is to introduce the type `?` to represent the *unknown type*. For example, the type `Int` $\to$ `?` is the type of a function which will output an element of unknown type when given an input of integer type. Types with unknown components stem from either explicit annotations or because they are deduced. For instance in

```
let foo (x : ?) = x + 1
let bar (x : ?) = x
let baz (x : Int) = bar(x)
```

it is natural to assign the type `? → Int` to `foo`, `? → ?` to `bar`, and `Int → ?` to `baz`. This is so because in gradual typing the types `Int` and `?` are *consistent*, that is, they are two types that at runtime *may* turn out to be the same (in particular because the `?` may turn out to be any type and, thus, `Int`).[1] A gradual type-checker checks that types are consistent rather than equal, which is why it allows to use expressions of type unknown ? where an expression of a static type such as Int is expected (e.g., the `x` in the body of `foo`) and use an expression of a static type such as Int where an expression of unknown type is expected (e.g., the `x` in the body of `baz`). The absence of type annotation can be either treated as an implicit `?` annotation (dynamic languages are a special case of it) or by using classic type reconstruction techniques (as in ML).

Cast Calculus. In practice, if an expression e of type `?` is used in a context that requires an integer, such as $e + 1$, the gradual type checker will consider that `?` is consistent with `Int`, and the expression $e + 1$ will type-check. But this might backfire, at runtime, if e evaluates to anything else than an integer. In order to ensure that this is not the case, the compiler will insert in the program safeguards that dynamically enforce type constraints on gradually-typed expressions. For instance, $e+1$ will be compiled into $e\langle\texttt{Int}\rangle+1$ where $e\langle\texttt{Int}\rangle$ is a type-cast expression that dynamically checks whether the result of e has type `Int`. In particular, `foo` will be compiled as `foo(x : ?) = (x⟨Int⟩ + 1)`. The target language of this compilation, with explicit type-casts, is called a *cast calculus*.
Let us recap the process with the following example:

```
let f (condition) (x : ?) =
  if condition then x + 1 else ¬x
```

This is a function that takes two arguments, `condition` and `x`, and returns either `x+1` if `condition` is `true`, or the negation of `x`, `¬x`, if `condition` is `false`. This code is typed with `Bool → ? → ?` by gradual type system of Siek and Vachharajani [24]. This knowledge is then used to insert dynamic type checks to ensure that the value bound to `x` will be, according to the case, an integer or a Boolean:

```
let f (condition) (x : ?) =
    if condition then x<Int> + 1 else ¬(x<Bool>)
```

[1]For example the three types of `foo`, `bar`, and `baz` are pairwise consistent, but the type `Bool → ?` is consistent only with the first two.

But looking again at this example, we see that its derived type `Bool → ? → ?` is quite imprecise. For example, if we pass a value that is neither an integer nor a Boolean (e.g., a list) as the last argument `x` to `f`, then this application is well-typed due to the type-casts which enforces the type of `x`, even though the execution will always fail, independently of the value of `condition`. Likewise, the type gives no useful information about the result of `f`, even though it will clearly be either an integer or a Boolean, and nothing else. This problem can be solved by using more precise types and, in this case, set-theoretic types.

Set-theoretic types. Set-theoretic types include intersection types $t_1 \wedge t_2$, union types $t_1 \vee t_2$, and negation types $\neg t$. These constructors respect naive set intuitions, so that an expression of type t_1 and t_2 can be given the type $t_1 \wedge t_2$, etc. In our previous example, they allow the programmer to annotate the argument `x` more precisely:

```
let f (condition) (x :  (Int | Bool) & ?) =
    if condition then x<Int> + 1 else ¬(x<Bool>)
```

where "`|`" denotes union and "`&`" denotes intersection. The union `(Int | Bool)` indicates that a value of this type is *either* an integer *or* a Boolean, and the intersection indicates that `x` has *both* type `(Int | Bool)` *and* type `?`. Intuitively, this type annotation means that the function `f` accepts for `x` a value of any type (which is indicated by `?`), as long as this value is also either an integer *or* a Boolean. The use of the intersection of a union type with "?" to type a parameter corresponds to a programming style in which the programmer asks the system to *statically* enforce that the function will be applied only to arguments in the union type and delegates to the system any *dynamic* check regarding the use of the parameter in the body of the function. The system by Castagna et al. [7] is able to deduce for this code example the type:

```
Bool → ((Int | Bool) & ?) → (Int | Bool)
```

The return type is no longer gradual, thanks to set-theoretic types. This is a useful feature as more precision in types may be a source of optimizations. In this case, it might be crucial for the run-time type checker or the compiler to have the information that the result will be of type `(Int | Bool )`. However, the use of set-theoretic types brings new complexity issues when dealing with casts. In this example, it was easy to reconstruct the return type of `f` because the operators + and negation ¬ had simple static types: respectively, `Int×Int→Int` and `Bool→Bool`. But set-theoretic types encode complex function types whose return types are not obvious and need to be computed. For example, intersection types can be used to encode *ad-hoc polymorphism* (a.k.a., function overloading), where a piece of code acts on more than one type with different behavior in each case. See the type τ = `(Int → Int)` $\wedge$ `(Bool → Bool)`, which denotes functions that will return an integer if applied to an integer, and will return a Boolean if applied to a Boolean. It is possible to compute the domain of such a type (i.e., of all functions of this type), denoted by $\texttt{dom}\,\tau = \texttt{Int} \vee \texttt{Bool}$, that is the union of all the possible input types. But what is the precise return type of such a function? It depends on what the argument of such a function is: either an integer or a Boolean—or both, denoted by the union type $\texttt{Int} \vee \texttt{Bool}$. Therefore, an operator on types needs to be defined which we denote by $\circ$. More precisely, we denote by $\tau_1 \circ \tau_2$ the type of an application of a function of type τ_1 to an argument of type τ_2.

In the example with $\tau = (\mathtt{Int} \to \mathtt{Int}) \wedge (\mathtt{Bool} \to \mathtt{Bool})$, it gives $\tau \circ \mathtt{Int} = \mathtt{Int}$, $\tau \circ \mathtt{Bool} = \mathtt{Bool}$, and $\tau \circ (\mathtt{Int} \vee \mathtt{Bool}) = \mathtt{Int} \vee \mathtt{Bool}$. The execution of a cast calculus with set-theoretic types requires the operations of domain and result to be computed at run-time (e.g., when an unknown function is cast to the type τ above), likewise for the subtyping relation. In this paper we show how to limit the impact of these computations on performance.

Overview. In Section 2, we start by presenting the intuitions behind the gradual set-theoretic types, we describe the main challenges to obtain a space efficient implementation, and we formally describe the cast language implemented by our virtual machine: its syntax, dynamic, and static semantics. Section 3 describes our virtual machine and how to compile the cast language into its bytecode. Section 4 studies the space efficiency of our machine and Section 5 its time efficiency via a chosen set of benchmarks. Discussion of related work of future extension and a conclusion end the presentation. For space reasons, several definition and all the proofs are relegated to an appendix available online. The source code of our virtual machine and of the benchmarks is available at https://github.com/gliboc/cast-machine.

2 CAST LANGUAGE

In this section we present the cast language implemented by our virtual machine. The syntax is that of the cast language by Castagna et al. [7] tweaked for efficient implementation. In particular, we modify the syntax of casts to remove unessential parts and to memoize the information about the domain of function types (for the reasons we explain in Section 2.2). So our casts will be of the form $\langle\tau_1\rangle^{\tilde\tau_2}$, where $\tilde\tau_2$ is used to store information about the domain of τ_1 when this is a function type and it is $\bot$ otherwise. We do not explain how to compile a gradually-typed program into a program of the cast language since this is essentially[2] the same as in [7]. To define the cast language we have to introduce the gradual types and their subtyping relation (the *consistency* relation is factored out by the compilation), the syntax of the calculus and its dynamic and static semantics, as we do next.

2.1 Gradual set-theoretic types

Let b range over a set $\mathscr{B}$ of *basic types* (e.g., $\mathscr{B} = \{\mathtt{Int}, \mathtt{Bool}\}$). Following the approach of Castagna et al. [7], we define both gradual and static (a.k.a. non-gradual) set-theoretic types as the terms produced coinductively by the following grammars

$$\begin{array}{rl} \text{static types} & t ::= b \mid t \times t \mid t \to t \mid t \vee t \mid \neg t \mid \mathbb{0} \\ \text{gradual types} & \tau ::= ? \mid b \mid \tau \times \tau \mid \tau \to \tau \mid \tau \vee \tau \mid \neg\tau \mid \mathbb{0} \end{array}$$

and that satisfy the following conditions:

- *(regularity)* the term has a finite number of different sub-terms;
- *(contractivity)* every infinite branch of a type contains an infinite number of occurrences of products or arrows.

We refer the reader to [13] and [7] for more explanation about these restrictions. We introduce the following abbreviations for gradual types: $\tau_1 \wedge \tau_2 \overset{\text{def}}{=} \neg(\neg\tau_1 \vee \neg\tau_2)$, $\tau_1 \setminus \tau_2 \overset{\text{def}}{=} \tau_1 \wedge \neg\tau_2$, $\mathbb{1} \overset{\text{def}}{=} \neg\mathbb{0}$, and likewise for static types. We refer to b, $\times$, and $\to$ as *type constructors* and to $\vee$, $\wedge$, $\neg$, and $\setminus$ as *type connectives.*

Type connectives are only truly meaningful in presence of a subtyping relation: union and intersection can respectively be defined as the least upper bound and the greatest lower bound for this relation, while $\mathbb{0}$ and $\mathbb{1}$ can be respectively defined as the extrema of the lattice. In this work, we choose to reuse the semantic subtyping relation defined on both static and gradual set-theoretic types by Castagna et al. [7]. This definition being complex, we omit the details in this paper, and instead give the following sound but not complete set of rules for intuition:

$$\frac{}{\tau \le \tau} \qquad \frac{}{\tau \le \mathbb{1}} \qquad \frac{\tau_1 \le \tau_2}{\neg\tau_2 \le \neg\tau_1} \qquad \frac{\tau_1 \le \tau_3 \quad \tau_2 \le \tau_4}{\tau_1 \times \tau_2 \le \tau_3 \times \tau_4}$$

$$\frac{\tau_3 \le \tau_1 \quad \tau_2 \le \tau_4}{\tau_1 \to \tau_2 \le \tau_3 \to \tau_4} \qquad \frac{\tau \le \tau_1 \quad \tau \le \tau_2}{\tau \le \tau_1 \wedge \tau_2} \qquad \frac{\tau_1 \le \tau \quad \tau_2 \le \tau}{\tau_1 \vee \tau_2 \le \tau}$$

In particular, ? is only a subtype of itself and of $\mathbb{1}$, and the same holds for $\neg$?.

2.2 Space Efficiency

There are two major problems one must solve when defining a space-efficient cast language with set-theoretic types.

Function Application. The first problem concerns the application of cast functions (i.e., of functions that have a cast applied to them). This issue comes from the fact that checking the type of a function against a cast is highly impractical. Consider for example:

```
let f (x : ?) = if x = 42 then 42 else true
```

This function can be given the dynamic type $? \to ?$. Since, intuitively, ? stands for any type (or the absence of type), this function can be cast to type $\mathtt{Int} \to \mathtt{Int}$. Moreover, it is clear that, for a certain value of its parameter ($x = 42$), this function returns an integer. Therefore, there are some execution contexts where casting this function to $\mathtt{Int} \to \mathtt{Int}$ is safe and will not produce a runtime error. However, it is also clear that this function *cannot be given* type $\mathtt{Int} \to \mathtt{Int}$ since it can also return a Boolean, and in some execution contexts, such a cast would fail.

The solution to decide whether a cast function such as $\mathtt{f}\langle\mathtt{Int} \to \mathtt{Int}\rangle$ should produce an error is to delay the evaluation of casts after the application of the function, as is usually done in the gradual typing literature [26]. The basic idea is to define a reduction rule for cast functions that resembles the following one, where $e\langle\tau\rangle$ denotes a cast of the expression e to the type τ and v denote values:

$$[\beta_{\langle\rangle}] \qquad v_1\langle\tau_1 \to \tau_2\rangle\, v_2 \quad\longrightarrow\quad (v_1\ v_2\langle\tau_1\rangle)\langle\tau_2\rangle$$

The presence of set-theoretic types make this problem much more difficult however, because casts applied to functions can contain arbitrarily complex types (intersections of unions of arrows). Hence, we need to use operators on types to compute the domain and the result type of an application. Using the operators outlined in the introduction and formally defined in [5], the rule becomes:

$$[\beta_{\langle\rangle}] \qquad v_1\langle\tau\rangle\, v_2 \quad\longrightarrow\quad (v_1\ v_2\langle\mathsf{dom}(\tau)\rangle)\langle\tau \circ \mathtt{type}(v_2)\rangle$$

where $\mathsf{dom}(\tau)$ computes the domain of τ, and $\tau \circ \tau'$ computes the result type of the application of a function of type τ to an argument of type τ', and $\mathtt{type}(e)$ returns the type of the expression e.

[2] The only difference is that whenever the compilation in [7] produces a cast of the form $\langle\tau' \overset{\ell}{\Rightarrow} \tau\rangle$ here we produce the cast $\langle\tau\rangle^{\mathsf{dom}\,\tau}$

Cast Accumulation. The second problem concerns casts that accumulate and create sequences of growing length. When the cast is in tail position, it can blow up the return stack during execution. For example, consider the mutually recursive functions:

```
let rec odd : Int -> ? = fun n ->
    if n = 0 then false<?>
    else (even (n-1))<?>
and even : Int -> Bool = fun n ->
    if n = 0 then true
    else (odd (n-1))<Bool>
```

Originally, these functions are written by the programmer in a gradually-typed language without casts, which corresponds to the code above without the green parts (i.e., the casts). During type-checking, the type annotation on function odd, which is Int→?, drives the type-checker to add explicit type-casts that enforce ? to be the return type of odd. In the same way, in the body of even, the call to odd is considered to return an element of type ? and has to be cast to Bool. Now, a naive evaluation of odd 5 would yield:

```
odd 5  →  (even 4)<?>
       →  (odd 3)<Bool><?>
       →  (even 2)<?><Bool><?>
       →  (odd 1)<Bool><?><Bool><?>
       →  (even 0)<?><Bool><?><Bool><?>
```

This short evaluation sequence highlights the problem many cast semantics have, which is space-inefficiency due to the accumulation of casts, possibly breaking tail-recursion. A solution that is used by the Grift compiler [18] is to implement a calculus on these casts, called the coercion calculus, in order to allow the composition of casts. Space efficiency is then achieved with the reduction that takes a chain-cast expression $E\langle c_1\rangle\langle c_2\rangle$ and computes $E\langle c_1 \fatsemi c_2\rangle$ where the $\fatsemi$ operation normalizes the sequencing of coercions c_1 and c_2 into a bounded-size representation. This approach has been carried out in the context of a simply-typed language (with functions, products), but it does not easily transfer to set-theoretic types.
We argue that, in the absence of blame-tracking, set-theoretic types can easily solve this problem. Remark that, intuitively, if an expression can be successfully cast to τ and to τ', it means that it has both of these types or, equivalently, that it has type $\tau \wedge \tau'$. Therefore, an expression $E\,\langle\tau\rangle\,\langle\tau'\rangle$ can be "compressed" to $E\,\langle\tau\wedge\tau'\rangle$. This idea is at the core of our space-efficient cast language we present next.

2.3 Language syntax

The expressions of the cast language are defined as follows (where $x \in \mathsf{Var}$ ranges over variables and c over constants):

$$\begin{array}{lll} \mathsf{Expr} & E ::= & x \mid c \mid \mu^{\tau\to\tau} f\,x\,.\,E \mid E\,E \mid E\,\langle\tau\rangle^{\tilde\tau} \\ & & \mid \texttt{let}\ x = E\ \texttt{in}\ E \mid \texttt{if}\ E\,E\,E \mid (E,E) \mid \pi_i\,E \\ \mathsf{Types}^{\perp} & \tilde\tau ::= & \tau \mid \perp \end{array}$$

For the most part, this language is a standard λ-calculus with constants, pairs (E,E), projections $\pi_i\,E$ (with $i \in \{1,2\}$), and a let construct. $(\mu^{\tau_1\to\tau_2} f\,x\,.\,E)$ stands for an abstraction explicitly annotated with type $\tau_1 \to \tau_2$, which uses f as a recursive binder. Intuitively, the application of such a function to a value v (values are defined in the next section) reduces to $E\,[f := \mu^{\tau_1\to\tau_2} f\,x\,.\,E]\,[x := v]$. More importantly, the construct $E\,\langle\tau\rangle^{\tilde\tau}$ stands for an expression that is cast to type τ, and that can safely be applied to elements of type $\tilde\tau$, provided it is a function (otherwise, $\tilde\tau = \perp$). Leaving the exponent part aside, this is a fairly standard cast construct [23, 26] where only information about the target type is kept (we do not need the source type, nor do we handle blame). If E reduces to some constant c then $E\,\langle\tau\rangle^{\tilde\tau}$ reduces to c if c has type τ, and fails otherwise.
The need for the exponent part comes from the fact that, as we previously said, we merge successive casts together into a single one (in our implementation space depends more on the number of casts than on the form of their types). This operation is straightforward thanks to set-theoretic types: an expression $E\,\langle\tau\rangle\,\langle\tau'\rangle$ is "compressed" to $E\,\langle\tau\wedge\tau'\rangle$. However, for function casts, a different operation must be done on the domain of the cast. Using the semantics for cast applications we described before, an application $v\,\langle\texttt{Int}\to\texttt{Int}\rangle\,\langle ?\to ?\rangle\,\langle\texttt{Bool}\to\texttt{Bool}\rangle\,v'$ must never succeed, since this would result in the argument v' being cast from Bool to ? to Int before being passed to the function v, and no value can be of both types Bool and Int. However, if we were to merge these casts naively, we would obtain the application $v\,\langle(\texttt{Int}\to\texttt{Int})\wedge(?\to ?)\wedge(\texttt{Bool}\to\texttt{Bool})\rangle v'$. Since $(\texttt{Int}\to\texttt{Int})\wedge(?\to ?)\wedge(\texttt{Bool}\to\texttt{Bool})$ is a subtype of $(\texttt{Int}\vee ?\vee\texttt{Bool})\to(\texttt{Int}\vee ?\vee\texttt{Bool})$, then in particular this application will succeed whenever v' is of type either Int or Bool, which would be unsound. More formally, the issue is that taking the intersection of two function types increases their domains (the domain of an intersection being the union of the domains), while casting a function to two different types makes it less likely to succeed when applied. To compress casts by intersections, thus, we record casts on the domain of a function independently from its type, hence the exponent. In particular we now compress $E\,\langle\tau\rangle^{\tau_1}\,\langle\tau'\rangle^{\tau_2}$ into $E\,\langle\tau\wedge\tau'\rangle^{\tau_1\wedge\tau_2}$. Adding this information to casts, the previous application becomes $v\,\langle\texttt{Int}\to\texttt{Int}\rangle^{\texttt{Int}}\,\langle ?\to ?\rangle^{?}\,\langle\texttt{Bool}\to\texttt{Bool}\rangle^{\texttt{Bool}}\,v'$, which reduces to $v\,\langle(\texttt{Int}\to\texttt{Int})\wedge(?\to ?)\wedge(\texttt{Bool}\to\texttt{Bool})\rangle^{\texttt{Int}\wedge ?\wedge\texttt{Bool}} v'$. This expression will then always fail, since the type of the argument is now checked against the exponent, and $\texttt{Int}\wedge ?\wedge\texttt{Bool}$ is empty (since $\texttt{Int}\wedge\texttt{Bool}$ is so) and thus no value can have this type.

2.4 Big step semantics

We present the big step semantics of the cast calculus since it is closer to our our virtual machine (cf. Theorem 2). The (equivalent) small step semantics is used just to state the type safety property and omitted for space reasons (it can be found in Appendix **??**). First we define the syntax of values, for which we partition the set of constants (ranged over by c) in functional constants (i.e., constants with a function type) ranged over by $o \in$ Functionals and non functional ones ranged over by $k \in$ Constants:

$$\begin{array}{lll} \mathsf{Func} & u := & o \mid \mu^{\tau\to\tau} f\,x\,.\,E \\ \mathsf{Val} & v := & k \mid (v,v) \mid \left[u, \mathcal{E}, \langle\tau\rangle^{\tau}\right]^{m} \\ \mathsf{Mark} & m := & \star \mid \square \end{array}$$

The operator type[3] is defined on values $v \in \mathsf{Val}$ as:

$$\begin{array}{rl} \texttt{type}(\left[u, \mathcal{E}, \langle\tau_1\rangle^{\tau_2}\right]^{m}) & = \tau_1 \\ \texttt{type}\ k & = \mathbb{B}(k) \wedge ? \wedge \neg ? \\ \texttt{type}\,(v_1, v_2) & = \texttt{type}\ v_1 \times \texttt{type}\ v_2 \end{array}$$

[3]This operator is used to get the type of values that are used as arguments of an application in order to compute the result type of the application. For more details on the definition, especially in the case of constants, see the Section 2.5 (cf., rule [Const]).

where $\mathbb{B}$ maps non functional constants into their basic type. Our values include closures $\left[u, \mathcal{E}, \langle\tau\rangle^{\tau}\right]^{m}$, which are formed by a functional expression u, an environment $\mathcal{E}$ which is a finite function $\texttt{Var} \to \texttt{Val}$, a pair of types and, at the index, a mark m. This mark distinguishes between two cases: when $m = \square$, it means that the cast $\langle\tau\rangle^{\tau}$ is waiting to be applied to the function inside the closure; when $m = \star$, $\langle\tau\rangle^{\tau}$ is kept as a static type information regarding this function, but the closure will be applied without adding or enforcing type-casts. In other terms, $\square$-closures represent functions to which a cast is applied (they will be reduced by using $\beta_{\langle\rangle}$) while $\star$-closures are usual functions without any cast (they will be reduced by the usual β). We distinguished between functional constants o and constants k, because we want to have a support for pre-defined functions in our language (e.g., +, *incr*, …), and because casts behave differently on these two kinds of constants.
Closures are created by capturing the current environment and the type in the abstraction. At first, such a closure should not enforce any typing constraints, hence the $\star$ mark. Functional constants are wrapped into similar closure only with empty environments since they do not have free variables needing to be bound.

$$\frac{}{\mathcal{E} \vdash \mu^{\tau_1 \to \tau_2} f\, x\,.\, E \Rightarrow [\mu^{\tau_1 \to \tau_2} f\, x\,.\, E,\ \mathcal{E},\ \langle \tau_1 \to \tau_2 \rangle^{\tau_1}]^{\star}}$$

Then, the following rules handle function application. The first, $[\beta_\star]$, is a standard untyped function application that corresponds to the standard β-reduction. The second, $[\beta_\square]$, enforces cast constraints on the arguments and on the result of the application, which this time corresponds to the rule $[\beta_{\langle\rangle}]$ outlined earlier. We denote by $\mathcal{E}\{x := v\}$ a copy of $\mathcal{E}$ which outputs v when given x.

$$[\beta_\star]\ \frac{\begin{array}{c}\mathcal{E} \vdash E_1 \Rightarrow v_1 = [\mu^{\tau} f\, x\,.\, E,\ \mathcal{E}',\ \kappa]^{\star} \\ \mathcal{E} \vdash E_2 \Rightarrow v_2 \qquad \mathcal{E}'\{x := v_2\ f := v_1\} \vdash E \Rightarrow v\end{array}}{\mathcal{E} \vdash E_1\, E_2 \Rightarrow v}$$

$$[\beta_\square]\ \frac{\begin{array}{c}\mathcal{E} \vdash E_1 \Rightarrow [u,\ \mathcal{E}',\ \langle\tau_1\rangle^{\tau_2}]^{\square} \quad v_1 = [u,\ \mathcal{E}',\ \langle\tau_1\rangle^{\tau_2}]^{\star} \\ \mathcal{E} \vdash E_2 \Rightarrow v_2 \qquad \mathcal{E} \vdash v_2\langle\tau_2\rangle^{\mathrm{dom}\tau_2} \Rightarrow v_3 \\ \mathcal{E} \vdash (v_1\, v_3)\langle\tau_1 \circ \mathtt{type}(v_2)\rangle^{\mathrm{dom}\,(\tau_1 \circ \mathtt{type}(v_2))} \Rightarrow v\end{array}}{\mathcal{E} \vdash E_1\, E_2 \Rightarrow v}$$

Let us see how $[\beta_\square]$ works. When applying a closure $[u,\ \mathcal{E}',\ \langle\tau_1\rangle^{\tau_2}]^{\square}$, the index type τ_2 should already capture the type of the argument. Therefore, the argument E_2 is evaluated to a value v_2 and then cast to v_3 using τ_2. During that time, the type $\tau = \tau_1 \circ \mathtt{type}(v_2)$ is computed in order to best approximate the result type of the function call. Finally, $(v_1\, v_3)\langle\tau\rangle^{\mathrm{dom}\ \tau}$ can be evaluated, where v_1 is the closure of the beginning whose mark has been set to $\star$, meaning that the next reduction used will be the untyped one, $[\beta_\star]$.
The problem of accumulating casts is solved by systematically reducing such proxies both on constants and on closures. Reduction of casts on closures is achieved by the following rule:

$$\frac{\mathcal{E} \vdash E \Rightarrow [u,\ \mathcal{E}',\ \langle\tau_3\rangle^{\tau_4}]^{m}}{\mathcal{E} \vdash E\langle\tau_1\rangle^{\tau_2} \Rightarrow \left[u,\ \mathcal{E}',\ \langle\tau_1 \wedge \tau_3\rangle^{\tau_2 \wedge \tau_4}\right]^{\square}}$$

This rule uses intersection types to compress type-casts on closures. As we show in Section 4, this representation is bounded in space according to the number of type annotations in the source program. After computing the new type-cast, the closure is tagged with $\square$ in order to signify that it should be applied using rule $[\beta_\square]$, which enforces type constraints.

In order to reduce casts on constants, we consider the fact that the gradual part of a cast does not influence the result of the cast. Indeed, if we cast an integer to a gradual type, as in `42<Int∧?>` or `42<Bool∧?>`, the only parts of theses casts that have a consequence are `Int` or `Bool`, since `?` could represent anything. Therefore, we can erase `?` in both these casts, and re-apply the newly-obtained cast.[4] To this purpose we define the extrema of gradual types in order to be able to cast constants to gradual types.

Definition 1. **(Gradual Extrema)** *For every gradual type τ,*

- *the* gradual maximum $\tau^{\Uparrow}$ *is obtained by replacing every* covariant *occurrence of* ? *by* $\mathbb{1}$*, and every* contravariant *occurrence by* $\mathbb{0}$
- *the* gradual minimum $\tau^{\Downarrow}$ *is obtained by replacing every* contravariant *occurrence of* ? *by* $\mathbb{1}$*, and every* covariant *occurrence by* $\mathbb{0}$

This definition is made so that the type-cast from $\mathbb{B}(k)$ to τ_1 will succeed whenever $\mathbb{B}(k) \leq {\tau_1}^{\Uparrow}$—i.e., that it is not a problem to ignore the gradual part of the cast that is erased by taking the gradual extrema, since all constants are implicitly gradually typed (see Footnote 1). The condition resulting from, written $\mathbb{B}(k) \leq \tau_1^{\Uparrow}$, is implemented by the two following rules, which imply the full reduction of casts on constants:

$$\frac{\mathcal{E} \vdash E \Rightarrow k \quad \mathbb{B}(k) \leq \tau_1^{\Uparrow}}{\mathcal{E} \vdash E\langle\tau_1\rangle^{\tau_2} \Rightarrow k} \qquad \frac{\mathcal{E} \vdash E \Rightarrow k \quad \mathbb{B}(k) \not\leq \tau_1^{\Uparrow}}{\mathcal{E} \vdash E\langle\tau_1\rangle^{\tau_2} \Rightarrow \mathtt{Fail}}$$

The full set of rules for the big step semantics can be found in Figure 10 in the annexes.

2.5 Type System

Since a cast language is not meant to be used by a programmer, but rather by the compiler as an intermediate language, defining its type system is only necessary to prove the soundness of its semantics, which is the point of this section. The type systems uses most of the standard rules of a simply-typed lambda-calculus with pairs plus classic subsumption for subtyping:

$$[\textsc{Subsume}]\ \frac{\Gamma \vdash E : \tau'}{\Gamma \vdash E : \tau}\ \tau' \leq \tau$$

However, there are two major differences coming from the typing rules for non-functional constants and for casts, stated as follows:

$$[\textsc{Const}]\frac{}{\Gamma \vdash k : \mathbb{B}(k) \wedge ? \wedge \neg ?} \qquad [\textsc{Cast}]\frac{\Gamma \vdash E : \tau}{\Gamma \vdash E\langle\tau_1\rangle^{\tilde{\tau}_2} : \tau_1}$$

The rule for casts states that, since τ_1 is the most precise type that can be given to $E\langle\tau_1\rangle^{\tau_2}$ if the cast succeeds, we simply give this type to the expression, provided E is well-typed. Notice that both τ and $\tilde{\tau}_2$ are disregarded by the rule
The rule for constants is certainly the most bizarre one. Intuitively, this rule should be understood in this way: a constant k can be given type $\mathbb{B}(k)$ (which is its expected type), but can also be implicitely downcast to type ? and to ¬?. This comes from the fact that, in order to further optimize space consumption, we chose to remove all casts on non-functional constants. Consider for example the cast constant `42<?>`: this cast can never be the cause of an error, as a value can always be cast to the dynamic type, independently of its type. Therefore, such a cast can be removed without altering the

[4]This reduction erases the information given by having type-casts such as `?` on constants. Our type system will take that into account by adding a typing rule which allows the type-checker to implicitly type constants by "`?`" (and by "`¬?`" as well).

semantics of the program. However, this means in particular that `42<?>`, which is of type ?, reduces to `42`, which would intuitively be of type `Int`. Such a reduction would violate type preservation, hence the need for the rule [Const] which allows constant to be implicitely consider of type ?.
The part $\neg ?$ can be understood with the same reasoning, as ? and $\neg ?$ are actually semantically equivalent (but incomparable for subtyping). Indeed, since ? can intuitively represent any type τ, it can also represent the negation of any type $\neg\tau$. Therefore, $\neg ?$ can represent any double negation $\neg\neg\tau$, thus any type τ.
Using this type system, we can then prove for small step semantics the traditional lemmas from which the type safety theorem follows:

Lemma 1 (Preservation). *If $\Gamma \vdash E : \tau$ and $E \longrightarrow E'$, then $\Gamma \vdash E' : \tau$.*

Lemma 2 (Progress). *If $\Gamma \vdash E : \tau$ and E is closed, then either E is a value or $E \longrightarrow E'$ for some E'.*

Theorem 1 (Type safety). *If $\Gamma \vdash E : \tau$ and E is closed, then either E diverges, or $E \longrightarrow^\star v$ for some value v, or $E \longrightarrow^\star$ `Fail`.*

In order to prove that using $[\beta_\square]$ is practical, we now build a virtual machine that computes function calls space-efficiently.

3 VIRTUAL MACHINE

In this section we define our virtual machine and the compilation of the cast language into it. The main interest of our machine is that it provides a space-efficient implementation in presence of set-theoretic types. The two main problems of space-efficiency which we tackle are the efficient representation of casts and the compression of suite of casts (a sensitive problem in the presence of tail recursive calls of functions with cast).

3.1 Structure

A state of our machine is a 4-tuple (c, e, s, d) composed of a code pointer c, the current environment e, an operand stack s, and a control stack d (also called "dump") which handles return frames. The last three components of the machine are defined by the following grammar (the code c is defined below).

	$\mathsf{s} ::=$	$\varnothing \mid v\,.\,\mathsf{s} \mid \tilde\kappa\,.\,\mathsf{s} \mid \Omega\,.\,\mathsf{s}$	*Operand Stack*
	$\mathsf{d} ::=$	$\varnothing \mid \tilde\kappa\,.\,\mathsf{d} \mid (\mathsf{c},\mathsf{e})\,.\,\mathsf{d}$	*Control Stack*
	$\mathsf{e} ::=$	$\varnothing \mid v\,.\,\mathsf{e}$	*Environment*
MVal	$v ::=$	$c \mid (v,v) \mid [\mathsf{c},\mathsf{e},\kappa]^m$	*Machine Values*
	$\kappa ::=$	$\langle\tau\rangle^\tau$	*Type-casts*
	$\tilde\kappa ::=$	$\tau^{\tilde\tau}$	*Type Pairs*
	$m ::=$	$\star \mid \square$	*Reduction Marks*

The stack stores machine values v which encode the values of the cast language), type-casts κ, used to cast machine values that were pushed on the stack, and a failure mark Ω which denotes the failure of a cast. Machine values MVal include constants, pairs, and closures $[\mathsf{c},\mathsf{e},\kappa]^m$ containing a piece of code c, an environment e, a type-cast κ and a mark which indicates how to apply the closure, similarly as in Section 2.4 in the big-step semantics. Each machine value MVal is associated to its minimal (w.r.t. subtyping) type by the `type` operator defined as:

$$\begin{array}{lcl} \mathtt{type}\,([\mathsf{c},\mathsf{e},\langle\tau_1\rangle^{\tau_2}]^m) & = & \tau_1 \\ \mathtt{type}\,(c) & = & \mathbb{B}(c)\ \wedge\ ?\wedge\neg ? \\ \mathtt{type}\,((v_1,v_2)) & = & \mathtt{type}(v_1)\times\mathtt{type}(v_2) \end{array}$$

As for the operational semantics in Section 2.4, the mark $\square$ in a closures indicates that this closure has a cast pending to be applied, while the $\star$ mark indicates that a closure does not require any casts to be applied. No closure can contain a cast of the form $\langle\tau\rangle^\perp$, which is a pattern that indicates a cast that may only succeed on constants, and which fails on functions. However this pattern, in the form of $\tilde\kappa$, can appear on both the operand and the control stack.
The code of the virtual machine is a suite of instructions defined as follows:

$\mathtt{c} ::=$	$\varnothing \mid \mathtt{instr}\,;\mathtt{c}$	
$\mathtt{instr} ::=$	$\mid$ push obj	*Stack push*
	$\mid$ app $\mid$ ret	*Function call*
	$\mid$ tap	*Tail* app
	$\mid$ tca $\tilde\kappa$	*Cast tail* app
	$\mid$ cast	*Type-cast*
	$\mid$ ifz (c, c)	*Conditional*
	$\mid$ let $\mid$ end	*Let binders*
	$\mid$ pair	*Pairs*
$obj ::=$	$n \mid k \mid o \mid (\mathsf{c},\kappa) \mid \tilde\kappa$	$n \in$ Integers
$o ::=$	$\times \mid + \mid - \mid = \mid fst \mid snd$	

The push instruction injects identifiers, constants, closures, and types into the execution by pushing them onto the stack. Pair of types $\tilde\kappa$ are passed around in the stack and used by the tca instructions to create, compress, and apply casts at the execution.

3.2 Compilation

Our machine uses De Bruijn indices. To replace variables with indices, compilation uses a list of variables ρ and calls `lookup` $\rho\ x$ which returns the index of the first occurrence of x in ρ.
The compilation process is decomposed into two functions. The function $\mathcal{C}[\![]\!]_\rho$: Expr $\longrightarrow$ Bytes is the general one. It calls $\mathcal{T}[\![]\!]_\rho$ to compile expressions of that cast language that are in tail position, that is, the bodies of functions and the bodies of let expressions that are in tail position:

$$\begin{array}{lcl} \mathcal{C}[\![c]\!]_\rho & = & \mathsf{push}\ c \\ \mathcal{C}[\![x]\!]_\rho & = & \mathsf{push}\ (\mathtt{lookup}\ \rho\ x) \\ \mathcal{C}[\![\mu^{\tau_1\to\tau_2} f\,x\,.\,E]\!]_\rho & = & \mathsf{push}\ \left(\mathcal{T}[\![E]\!]_{f.x.\rho},\ \langle\tau_1\to\tau_2\rangle^{\tau_1}\right) \\ \mathcal{C}[\![E\,\langle\tau_1\rangle^{\tau_2}]\!]_\rho & = & \mathsf{push}\ \langle\tau_1\rangle^{\tau_2}\,;\,\mathcal{C}[\![E]\!]_\rho\,;\,\mathsf{cast} \\ \mathcal{C}[\![E_1\,E_2]\!]_\rho & = & \mathcal{C}[\![E_1]\!]_\rho\,;\,\mathcal{C}[\![E_2]\!]_\rho\,;\,\mathsf{app} \\ \mathcal{C}[\![\mathtt{let}\ x = E_1\ \mathtt{in}\ E_2]\!]_\rho & = & \mathcal{C}[\![E_1]\!]_\rho\,;\,\mathsf{let}\,;\,\mathcal{C}[\![E_2]\!]_{x.\rho}\,;\,\mathsf{end} \\ \mathcal{C}[\![(E_1,E_2)]\!]_\rho & = & \mathcal{C}[\![E_1]\!]_\rho\,;\,\mathcal{C}[\![E_2]\!]_\rho\,;\,\mathsf{pair} \\ \\ \mathcal{T}[\![E_1\,E_2]\!]_\rho & = & \mathcal{C}[\![E_1]\!]_\rho\,;\,\mathcal{C}[\![E_2]\!]_\rho\,;\,\mathsf{tap} \\ \mathcal{T}[\![(E_1\,E_2)\,\langle\tau_1\rangle^{\tau_2}]\!]_\rho & = & \mathcal{C}[\![E_1]\!]_\rho\,;\,\mathcal{C}[\![E_2]\!]_\rho\,;\,\mathsf{tca}\ \langle\tau_1\rangle^{\tau_2} \\ \mathcal{T}[\![\mathtt{let}\ x = E_1\ \mathtt{in}\ E_2]\!]_\rho & = & \mathcal{C}[\![E_1]\!]_\rho\,;\,\mathsf{let}\,;\,\mathcal{T}[\![E_2]\!]_{x.\rho} \\ \mathcal{T}[\![E]\!]_\rho & = & \mathcal{C}[\![E]\!]_\rho\,;\,\mathsf{ret} \end{array}$$

Our machine follows a classic eval-apply pattern with some specificities. In particular, the evaluation of a μ-abstraction pushes on the stack (see the definition of push further on) the closure which contains the body of the function (compiled for tail position), an environment containing both the parameter and the recursion variable, and a cast formed by the type annotation of the function and indexed with its domain. Applications are evaluated from left to right by the instruction app. When they are in tail position applications are compiled using the tap instruction that—contrary to app—does not save the calling context. If a cast is applied to a tail call, then the special tca instruction is used instead: acccording to the case, this instruction composes the cast on the tail call with the one on the top of the dump or the one in the callee, thus avoiding the problem of accumulation of casts we described before. Finally, casts that are not applied to tail calls are simply handled as plain operators whose arguments are first pushed on the stack and then evaluated by the execution of the cast instruction.

3.3 Transitions

3.3.1 Parameter functions. Several functions allow us to abstract functionalities of the virtual machine. The $\mathsf{push_e}(obj, \mathsf{s})$ function handles adding elements to the stack.

$$\begin{array}{ll}
\mathsf{push_e}(n, \mathsf{s}) & = \mathsf{e}(n).\,\mathsf{s} \\
\mathsf{push_e}(k, \mathsf{s}) & = k.\,\mathsf{s} \\
\mathsf{push_e}((\mathsf{c}, \kappa), \mathsf{s}) & = [\mathsf{c}, \mathsf{e}, \kappa]^{\star}.\,\mathsf{s} \\
\mathsf{push_e}(\tilde{\kappa}, \mathsf{s}) & = \tilde{\kappa}\,.\,\mathsf{s}
\end{array}$$

The composition of two pairs of types is defined as the symmetric operator that satisfies:

$$\begin{array}{lllll}
\tau^{\perp} \fatsemi {\tau_1}^{\tilde{\tau}_2} & = & {\tau_1}^{\tilde{\tau}_2} \fatsemi \tau^{\perp} & = & (\tau \wedge \tau_1)^{\perp} \\
{\tau_1}^{\tau_2} \fatsemi {\tau_3}^{\tau_4} & = & (\tau_1 \wedge \tau_3)^{\tau_2 \wedge \tau_4} & &
\end{array}$$

The cast function implements the application of casts on constants or closures using intersections to compress successive applications of function casts.

$$\begin{array}{lll}
\mathsf{cast}\left(k,\, {\tau_1}^{\tilde{\tau}_2}\right) & = & k \quad \text{if } \mathbb{B}(k) \leq {\tau_1}^{\Uparrow} \\
\mathsf{cast}\left([\mathsf{c}, \mathsf{e}, \langle\tau_1\rangle^{\tau_2}]^{m},\, {\tau_3}^{\tilde{\tau}}\right) & = & [\mathsf{c}, \mathsf{e}, \langle\tau_1 \wedge \tau_3\rangle^{\tau_2 \wedge \tilde{\tau}}]^{\square} \quad \text{if } \tilde{\tau} \neq \perp \\
\mathsf{cast}\,(v,\, \tilde{\kappa}) & = & \Omega \quad \text{in all other cases}
\end{array}$$

The dump d can contain either call frames or type pairs, which is what allows the elimination of tail calls even with casts on them. During a tail call, this function accumulates casts on top of the dump stack, in order to apply the resulting cast when the tail calls are over and the result of the computation is returned.

$$\begin{array}{ll}
\mathsf{dump}\,(\tilde{\kappa}_1,\, \tilde{\kappa}_2\,.\,\mathsf{d}) & = (\tilde{\kappa}_1 \fatsemi \tilde{\kappa}_2).\,\mathsf{d} \\
\mathsf{dump}\,(\tilde{\kappa},\, (\mathsf{c}, \mathsf{e})\,.\,\mathsf{d}) & = \tilde{\kappa}\,.\,(\mathsf{c}, \mathsf{e})\,.\,\mathsf{d} \\
\mathsf{dump}\,(\tilde{\kappa},\, \varnothing) & = \tilde{\kappa}
\end{array}$$

The table in Figure 1 describes the complete set of transitions of the virtual machine. There are three kinds of reduction: the usual function application $[\beta_{\star}]$, tail function calls $[t\beta_{\star}]$, and cast tail function calls $[c\beta_{\star}]$. While the first two are standard, the third one mixes a tail function call and a delayed cast application. It works by pushing a type-cast on the dump to be applied later, while performing a tail function call. There exists a typed version of each of these reductions: $[\beta_{\square}]$, $[t\beta_{\square}]$ and $[c\beta_{\square}]$ which are performed when the closure applied has a type-cast on it. Finally, the return instruction ret is also performed by two transitions: the usual one $[R_{\star}]$, which replaces the current frame by the top frame on the dump; and a typed one $[R_{\square}]$, which applies the type-cast on top of the dump to the current value on top of the stack.

3.4 Example

Let us compute odd 5 from the odd/even example of Section 2.2. By inlining the function even into the definition of odd, we can compile odd into a piece of bytecode $\mathsf{c_{odd}}$, which is put inside the closure $[\mathsf{c_{odd}}, \mathsf{e}', \langle\mathtt{Int} \to ?\rangle^{\mathtt{Int}}]^{\star}$. Now, we want to compute:

$$C[\![\mathtt{odd\ 5}]\!]_{\rho} = \mathsf{push}\ odd\,;\mathsf{push}\ 5\,;\mathsf{app}$$

These several key points illustrate the most important transitions used to compute odd 5 (abstracting inessential details away, such as De Bruijn indexes and recursion variables):

- First, the function call odd 5 is handled by regular application:

$$\begin{array}{l}
\left(\mathsf{app}\,,\, \mathsf{e}\,,\, [\mathsf{c_{odd}}, \mathsf{e}', \langle\mathtt{Int} \to ?\rangle^{\mathtt{Int}}]^{\star}.\,5\,,\, \varnothing\right) \\
\quad \to \left(\mathsf{c_{odd}}\,,\, \{n := 5\}.\,\mathsf{e}'\,,\, \varnothing\,,\, (\varnothing, \mathsf{e})\right)
\end{array}$$

- Then, the code of odd is executed and, since $n = 5$ is not zero, the conditional branch for the code `(even (n-1))<?>` is chosen. This expression is in tail position *and* consists of a cast on a function call. Therefore, the $[c\beta_{\star}]$ transition is applied and adds a type-cast "?" on the dump stack. Then, a tail call tap is executed, that is, a function call that does not push the current frame on the dump stack.

$$\begin{array}{l}
\left(\mathsf{tca}\ ?\,,\, \{n := 5\}.\,\mathsf{e}'\,,\, [\mathsf{c_{even}}, \mathsf{e}'', \langle\mathtt{Int} \to ?\rangle^{\mathtt{Int}}]^{\star}.\,4\,,\, (\varnothing, \mathsf{e})\right) \\
\quad \to \left(\mathsf{tap}\,,\, \{n := 5\}.\,\mathsf{e}'\,,\, [\mathsf{c_{even}}, \mathsf{e}'', \langle\mathtt{Int} \to ?\rangle^{\mathtt{Int}}]^{\star}.\,4\,,\, ?\,.\,(\varnothing, \mathsf{e})\right) \\
\quad \to \left(\mathsf{c_{even}}\,,\, \{n := 4\}.\,\mathsf{e}''\,,\, \varnothing\,,\, ?\,.\,(\varnothing, \mathsf{e})\right)
\end{array}$$

- The execution of $\mathsf{c_{even}}$ is similar to the one of $\mathsf{c_{odd}}$, and adds to the dump the typecast Bool which *composes* with the cast ? already on the top of the dump, resulting in the following:

$$\to \left(\mathsf{c_{odd}}\,,\, \{n := 3\}.\,\mathsf{e}'\,,\, \varnothing\,,\, (\mathtt{Bool} \wedge ?)\,.\,(\varnothing, \mathsf{e})\right)$$

- These cast tail calls keep decreasing the value of n, building up a *single* type-cast on top of the dump, until `even 0` is called, which returns true. But since a type-cast has been put on the dump stack, the result true must pass this type-cast before being returned, which yields the following final execution:

$$\begin{array}{l}
\left(\mathsf{push\ true}\,;\mathsf{ret}\,,\, \{n := 0\}.\,\mathsf{e}''\,,\, \varnothing\,,\, (\mathtt{Bool} \wedge ?)\,.\,(\varnothing, \mathsf{e})\right) \\
\quad \to \left(\mathsf{ret}\,,\, \{n := 0\}.\,\mathsf{e}''\,,\, \mathsf{true}\,,\, (\mathtt{Bool} \wedge ?)\,.\,(\varnothing, \mathsf{e})\right) \\
\quad \to \left(\mathsf{cast}\,;\mathsf{ret}\,,\, \{n := 0\}.\,\mathsf{e}''\,,\, (\mathtt{Bool} \wedge ?)\,.\,\mathsf{true}\,,\, (\varnothing, \mathsf{e})\right) \\
\quad \to \left(\mathsf{ret}\,,\, \{n := 0\}.\,\mathsf{e}''\,,\, \mathsf{true}\,,\, (\varnothing, \mathsf{e})\right) \\
\quad \to \left(\varnothing\,,\, \mathsf{e}\,,\, \mathsf{true}\,,\, \varnothing\right)
\end{array}$$

This example shows how the mechanism for handling casts on tail calls work in our virtual machine, by compressing type-casts on the dump stack using intersection types. Next we prove that this mechanism respects the semantics of our language.
We denote by $[\![\ ]\!]$ the function that maps the cast language values v into the corresponding machine values v, and extend it to environments $[\![\mathcal{E}]\!]$ pointwise. The link between the big step semantics and the virtual machine, is stated by the following theorem.

Theorem 2 (Soundness). *For every term E and environment $\mathcal{E}$, if $\mathcal{E} \vdash E \Rightarrow v$, then* $\forall(\mathsf{c}, \mathsf{s}, \mathsf{d})\ (C[\![E]\!]\,;\mathsf{c}, [\![\mathcal{E}]\!]\,, \mathsf{s}, \mathsf{d}) \to^{\star} (\mathsf{c}, [\![\mathcal{E}]\!], [\![v]\!].\mathsf{s}, \mathsf{d})$

3.5 Symbolic casts

A drawback of this virtual machine is that it relies on high-order operations on types: dom, ∘, ∧ (see the three $\_\beta_{\square}$ rules in Figure 1). Thank to the representation of types we describe in Section 4.1.1, intersection are less costly than the first two, since they consist of a simple merge of Binary Decision Diagrams (BDDs). This is why we

	BEFORE			AFTER			
SP	push obj ; c			c			
$\rightarrowtriangle$	e	s	d	e	$\mathsf{push}_{\mathsf{e}}(obj, \mathsf{s})$	d	
$\beta_\star$	app ; c			c'			
$\rightarrowtriangle$	e	$v.[\mathsf{c}', \mathsf{e}', \kappa]^\star.\,\mathsf{s}$	d	$[\mathsf{c}', \mathsf{e}', \kappa]^\star.v.\,\mathsf{e}'$	s	(c, e). d	
$\beta_\square$	app ; c			cast ; app ; cast ; c			$\tau = \tau_1 \circ \text{type } v$
$\rightarrowtriangle$	e	$v.[\mathsf{c}', \mathsf{e}', \langle\tau_1\rangle^{\tau_2}]^\square.\,\mathsf{s}$	d	e	$v.{\tau_2}^{\mathsf{dom}\ \tau_2}.\,[\mathsf{c}', \mathsf{e}', \langle\tau_1\rangle^{\tau_2}]^\star.\tau^{\mathsf{dom}\tau}.\,\mathsf{s}$	d	
$t\beta_\star$	tap ; c			c'			
$\rightarrowtriangle$	e	$v.[\mathsf{c}', \mathsf{e}', \kappa]^\star.\,\mathsf{s}$	d	$[\mathsf{c}', \mathsf{e}', \kappa]^\star.v.\,\mathsf{e}'$	s	d	
$t\beta_\square$	tap ; c			cast ; tca $\tau^{\mathsf{dom}\ \tau}$; c			$\tau = \tau_1 \circ \text{type } v$
$\rightarrowtriangle$	e	$v.[\mathsf{c}', \mathsf{e}', \langle\tau_1\rangle^{\tau_2}]^\square.\,\mathsf{s}$	d	e	$v.\,{\tau_2}^{\mathsf{dom}\ \tau_2}.\,[\mathsf{c}', \mathsf{e}', \langle\tau_1\rangle^{\tau_2}]^\star.\,\mathsf{s}$	d	
R$_\star$	ret ; c			c'			
$\rightarrowtriangle$	e	$v.\,\mathsf{s}$	$(\mathsf{c}',\mathsf{e}').\mathsf{d}$	e'	$v.\,\mathsf{s}$	d	
R$_\square$	ret ; c			cast ; ret ; c			
$\rightarrowtriangle$	e	$v.\,\mathsf{s}$	$\tilde\kappa.\mathsf{d}$	e	$v.\,\tilde\kappa\,.\,\mathsf{s}$	d	
$c\beta_\star$	tca $\tilde\kappa$; c			tap ; c			
$\rightarrowtriangle$	e	$v.[\mathsf{c}', \mathsf{e}', \kappa]^\star.\,\mathsf{s}$	d	e	$v.[\mathsf{c}', \mathsf{e}', \kappa]^\star.\,\mathsf{s}$	dump $(\tilde\kappa, \mathsf{d})$	
$c\beta_\square$	tca $\tilde\kappa$; c			cast ; tca $\left(\tilde\kappa \mathbin{⨟} \tau^{\mathsf{dom}\ \tau}\right)$; c			$\tau = \tau_1 \circ \text{type } v$
$\rightarrowtriangle$	e	$v.[\mathsf{c}', \mathsf{e}', \langle\tau_1\rangle^{\tau_2}]^\square.\,\mathsf{s}$	d	e	$v.{\tau_2}^{\mathsf{dom}\ \tau_2}.\,[\mathsf{c}', \mathsf{e}', \langle\tau_1\rangle^{\tau_2}]^\star.\,\mathsf{s}$	d	
C$_\perp$	cast ; c			$\varnothing$			$\mathtt{cast}\,(v, \tilde\kappa) = \Omega$
$\rightarrowtriangle$	e	$v.\ \tilde\kappa\ .\,\mathsf{s}$	d	$\varnothing$	$\Omega\,.\,\mathsf{s}$	(c, e). d	
C	cast ; c			c			$\mathtt{cast}\,(v, \tilde\kappa) = v'$
$\rightarrowtriangle$	e	$v.\ \tilde\kappa\ .\,\mathsf{s}$	d	e	$v'\,.\,\mathsf{s}$	d	
LET	let ; c			c			
$\rightarrowtriangle$	e	$v.\,\mathsf{s}$	d	$v.\,\mathsf{e}$	s	d	
END	end ; c			c			
$\rightarrowtriangle$	$v.\,\mathsf{e}$	s	d	e	s	d	

Fig. 1: Transitions of the virtual machine

concentrate on dom and $\circ$ and explore the possibility of delaying their application in a symbolic structure for casts.
This yields a variant of the virtual machine, which differs from the first in that it replaces pairs of types $\tilde\kappa$ by symbolic casts Σ:

$$
\begin{array}{lrl}
\Sigma\text{Bytes} & \mathsf{c} ::= & \varnothing \mid \mathtt{instr}\,;\mathsf{c} \\
 & \Sigma ::= & \tilde\kappa \mid \mathsf{dom}\ \Sigma \mid \mathsf{app}_\tau(\Sigma) \\
 & \mathtt{instr} := & \ldots \mid \mathsf{tca}\,\Sigma \mid \ldots
\end{array}
$$

Symbolic casts encode in their structure the calls to domain dom and cast composition $\circ$, and they are lazily evaluated by eval:

$$
\begin{array}{lll}
\mathtt{eval}\,(\mathsf{dom}\ \Sigma) & = \tilde\tau_2^{\,\mathsf{dom}\ \tilde\tau_2} & \text{with } \mathtt{eval}\,(\Sigma) = {\tau_1}^{\tilde\tau_2} \\
\mathtt{eval}\,\left(\mathsf{app}_\tau\,\Sigma\right) & = {\tau_r}^{\mathsf{dom}\ \tau_r} & \text{with } \mathtt{eval}\,(\Sigma) = {\tau_1}^{\tilde\tau_2} \\
 & & \quad\text{and } \tau_r = \tau_1 \circ \tau
\end{array}
$$

Structures. The structures of the machine are slightly different as well, as again we replace pairs of types $\tilde\kappa$ with symbolic casts Σ.

$$
\begin{array}{lrll}
\Sigma\text{Val} & v ::= & c \mid (v, v) \mid [\mathsf{c}, \mathsf{e}, \Sigma]^m & \textit{Machine values} \\
 & \mathsf{s} ::= & \varnothing \mid v\,.\,\mathsf{s} \mid \Sigma.\,\mathsf{s} \mid \Omega\,.\,\mathsf{s} & \textit{Operand stack} \\
 & \mathsf{d} ::= & \varnothing \mid \Sigma.\,\mathsf{d} \mid (\mathsf{c}, \mathsf{e}).\,\mathsf{d} & \textit{Control stack}
\end{array}
$$

Parameter functions. Since intersections are not symbolic operations, the composition of two symbolic pairs Σ_1, Σ_2 is defined as:

$$
\begin{array}{lcl}
\tau^\perp \mathbin{⨟} {\tau_1}^{\tilde\tau_2} & = & {\tau_1}^{\tilde\tau_2} \mathbin{⨟} \tau^\perp \;=\; (\tau \wedge \tau_1)^\perp \\
{\tau_1}^{\tau_2} \mathbin{⨟} {\tau_3}^{\tau_4} & = & (\tau_1 \wedge \tau_3)^{\tau_2 \wedge \tau_4} \\
\Sigma_1 \mathbin{⨟} \Sigma_2 & = & \mathtt{eval}\,(\Sigma_1) \mathbin{⨟} \mathtt{eval}\,(\Sigma_2)
\end{array}
$$

The transition rules of this virtual machine with symbolic casts stay the same, except for the three rules described in Figure 2.

4 SPACE EFFICIENCY

In this section we study the space efficiency of our virtual machine. There are two space-related problems to be considered: (1) the memory blueprint of casts created during the execution (2) the size of the structures (the three stacks for control, operands, and environment) during the execution.

4.1 Cast representation and compression

Our operational semantics uses intersection types to compress types, and the type operators (dom, $\circ$) to build new casts. To achieve space efficiency, we need to show that the representations of these type-casts are bounded in size. For this aspect of the space efficiency we can actually provide a formal proof based on the cast language itself. Let $|E|$ be the cast language expression obtained from erasing all casts from E. We have

Theorem 3. *For each program E there exists a constant factor c such that for all E', if $E \longrightarrow^\star E'$, then* $\mathsf{size}(E') \leq c \cdot \mathsf{size}(|E'|)$.

where size is a function defined on the cast language which measures the size of the representation of an expression. This theorem

$\beta_\square$		app ; c			cast ; app ; cast ; c	
$\rightarrowtriangle$	e	$\nu.[c', e', \Sigma]^\square.\, s$	d	e	$\nu.\Sigma_d.\, [c', e', \Sigma]^\star.\Sigma_r.\, s$	d
$t\beta_\square$		tap ; c			cast ; tca Σ_r; c	
$\rightarrowtriangle$	e	$\nu.[c', e', \Sigma]^\square.\, s$	d	e	$\nu.\, \Sigma_d.\, [c', e', \Sigma]^\star.\, s$	d
$c\beta_\square$		tca Σ_1 ; c			cast ; tca $(\Sigma_1 \fatsemi \Sigma_r)$; c	
$\rightarrowtriangle$	e	$\nu.[c', e', \Sigma]^\square.\, s$	d	e	$\nu.\Sigma_d.\, [c', e', \Sigma]^\star.\, s$	d

Fig. 2: Modified transitions of the virtual machine with symbolic casts (with $\Sigma_r = \mathsf{app}_{\text{type } \nu}(\Sigma)$ and $\Sigma_d = \mathsf{dom}(\Sigma)$)

states that the space required for the execution of a program which uses the type annotations of gradual typing is bounded, as it stays within a constant factor of the space required to execute the same untyped program. In other terms, the space used by casts during the execution is bounded by a factor constant at all times. To see this we need to be more precise and define the size required to represent a program:

$$\begin{aligned}
\text{size}\, x \;=\; \text{size}\, c \quad &= 1 \\
\text{size}\,(E_1\, E_2) = \text{size}\,(E_1, E_2) \quad &= 1 + \text{size}\, E_1 + \text{size}\, E_2 \\
&\;\vdots \\
\text{size}\,(\pi_i\, E) \quad &= 1 + \text{size}\, E \\
\textbf{size}\;(\mathrm{E}\langle\tau_1\rangle^{\tilde{\tau}_2}) \quad &= \mathbf{1 + size\, E + size}\, \tau_1 + \mathbf{size}\;\tilde{\tau}_2
\end{aligned}$$

Establishing a bound on the size of the representation of types used during execution—which is the difference in size between untyped and typed versions of a program—is key to the proof of this theorem. The existence of this bound is due to the fact that in our machine types are represented by *Binary Decision Diagrams* (BDD). So in the definition above the size of a type is the size of the BDD representing it.

4.1.1 Binary decision diagrams. The subtyping algorithm for set-theoretic types works with types in disjunctive normal forms, which are best represented by Boolean functions [4]. It follows that the classic representation structure for set-theoretic types is BDDs. We will now present the representation of set-theoretic types which is used in CDuce, and therefore in the implementation of our virtual machine. First, we introduce an equivalent definition for types based on *atoms*. Let b range over a set $\mathscr{B}$ of basic types. Gradual set-theoretic types are the possibly infinite terms produced coinductively by

$$\begin{array}{lll}
\text{Atoms} & a ::= & b \mid\; ? \;\mid\; \tau \to \tau \\
\text{Types} & \tau ::= & a \mid \tau \vee \tau \mid \neg\tau \mid \mathbb{0}
\end{array}$$

with the same condition and abbreviation as in Section 2.1. Frisch et al. [13] proved that every type is equivalent to (i.e., denotes the same set of values as) a type in Disjunctive Normal Form:

$$\bigvee_{i \in I} \left(\bigwedge_{p \in P_i} a_p \wedge \bigwedge_{n \in N_i} \neg a_n \right) \qquad (1)$$

BDDs are defined by the grammar $B ::= \bot \mid \top \mid a?B : B$ and have the following interpretation:

$$\begin{aligned}
[\![\top]\!] \quad &= \mathbb{1} \\
[\![\bot]\!] \quad &= \mathbb{0} \\
[\![a?B_1 : B_2]\!] \quad &= (a \wedge [\![B_1]\!]) \vee (\neg a \wedge [\![B_2]\!])
\end{aligned}$$

which allows to convert any BDD to a type in disjunctive normal form—see Figure 3 for an example. To ensure that the atoms occurring on a path are distinct, a total order is defined on the atoms

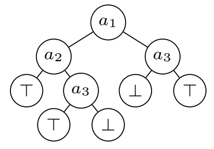

Fig. 3: BDD for $(a_1 \wedge a_2) \vee (a_1 \wedge \neg a_2 \wedge a_3) \vee (\neg a_1 \wedge \neg a_3)$

which imposes that on every path the order of the labels strictly increases. Besides, hash consing is used for atoms and, thus, several occurrence of the same atom (e.g., a_3 in Fig. 3) share the same representation. Hence the depth of a BDD is upper-bounded by the number of atoms available to build it. This yields the result:

Lemma 3. *For B a BDD, let $\mathcal{A} = \{a \in B\}$ be the set of distinct atoms in B, and $\alpha = |\mathcal{A}|$, we have*

$$\text{size}(B) \leq 2^\alpha \log_2 \alpha + \sum_{a \in \mathcal{A}} \text{size}\, a \overset{\text{def}}{=} C_{\mathcal{A}}$$

In this formula, we consider the maximum number of nodes of a tree of depth α, which is 2^α. All the distinct atoms are stored separately using size $\sum_{a \in \mathcal{A}}$ size a, each node of the tree being a reference to a stored atom (*i.e.* of size $\log_2 \alpha$).[5] What we now prove is that, in fact, the set of distinct atoms that can be used to build casts for a given program is fixed and does not vary during the execution. An initial cast expression E contains a bounded amount of type annotations and type-casts, from which only a bounded amount of atoms can be extracted. And because the creation of type-casts in the operational semantics is conservative in that it never creates any new atom, we can bound the size of any BDD-represented type-cast during the execution of a program by $C_{\mathcal{A}}$, with $\mathcal{A}$ the set of distinct atoms initially derived from the program. Formally:

Theorem 4. *Let E be a cast expression. There exists a finite set of atoms $\mathcal{A}$ such that for all E', if $E \longrightarrow^\star E'$, then every type-cast occurring E' is represented by a BDD using exclusively atoms from $\mathcal{A}$.*

Corollary 5. *Any type-cast $\langle\tau_1\rangle^{\tilde{\tau}_2}$ in E' is bounded in size, by* size $\langle\tau_1\rangle^{\tilde{\tau}_2} \leq 2C_{\mathcal{A}}$

Theorem 4 comes from the fact that the operations on types used at run-time do not create new atoms. To create new casts the machine uses type intersection $\wedge$, and the domain and result operators dom, $\circ$. Let us describe how each operation handles atoms:

Intersections: consider the intersection of two BDDs. Let B_1 and B_2 denote generic BBDs, $B_1 = a_1?C_1 : D_1$, $B_2 = a_2?C_2 : D_2$. Intersections of BDDs are defined as follows:

$$\bot \wedge B = B \wedge \bot = \bot \qquad \top \wedge B = B \wedge \top = B$$

[5]This is a conservative approximation: the actual representation of types is a record of several BDDs, one for each type constructor, as described in Section 4.3 of [4].

bench	fun. calls	app (%)	tap(%)	tca (%)
sieve	$2.6 \cdot 10^6$	59.6%	20.8%	19.6%
odd-even	10^7	$\sim$ 0%	0%	$\sim$ 99%

Fig. 4: Benchmark of Space Consumption (Tail Calls)

$$B_1 \wedge B_2 = \begin{cases} a_1 ? C_1 \wedge C_2 : D_1 \wedge D_2 & \text{for } a_1 = a_2 \\ a_1 ? C_1 \wedge B_2 : D_1 \wedge B_2 & \text{for } a_1 < a_2 \\ a_2 ? B_1 \wedge C_2 : B_1 \wedge D_2 & \text{for } a_1 > a_2 \end{cases}$$

This definition makes it clear that the atoms of $B_1 \wedge B_2$ are included in the union of the atoms of B_1 and B_2, so no new atom is created. The same property holds for union and negation.

Domain operator: if τ is a function type then it can be represented in disjunctive normal form as [13]:

$$\tau = \bigvee_{i \in I} \left(\bigwedge_{p \in P_i} (\sigma_p \to \tau_p) \wedge \bigwedge_{n \in N_i} \neg(\sigma_n \to \tau_n) \right)$$

and its domain is defined as $\mathsf{dom}\,\tau = \bigwedge_{i \in I} \bigvee_{p \in P_i} \sigma_p$ (see [13]).

Since unions and intersections do no create new atoms, then it is enough to include in the initial set of atoms $\mathcal{A}$ all the atoms used in the domain σ_p of every single function atom $a = \sigma_p \to \tau_p$ in E.

Result operator: for τ the function type defined above, the *codomain* is defined as

$$\mathsf{cod}\,\tau = \bigvee_{i \in I} \bigvee_{p \in P_i} \tau_p$$

We know that, for any type σ, the result types $\tau \circ \sigma$ will be formed using the atoms τ_p of $\mathsf{cod}\,\tau$ — see the annexes for a formal definition of $\tau \circ \sigma$ which makes this more explicit. Therefore, in the same way as we did for the domain, adding all the atoms of the codomain τ_p of every single function atom $a = \sigma_p \to \tau_p$ in E allows the conservation of atoms when using the result operator $\circ$.

This principle of conservation allows us to conclude with Theorem 4; and then, Corollary 5 yields a bound on the size of every type-cast during the execution of a program E. Since our semantics does not create chains of casts during execution, this allow us to conclude that the space overhead of casts during execution is indeed, bounded, as stated in Theorem 3. For a formal proof of these claims, the reader can refer to Section F in the annexes.

4.2 Tail recursion and stack space efficiency

The space-efficiency of this machine comes from three sources:

- a bounded representation for type-casts
- a semantics which reduces all chains of casts into a single one
- an instruction to handle casts on tail function calls

We formally proved in the previous section the first two points. We studied the third on examples and by comparisons with other works, but we did not provide a formal characterization of this aspect. Instead, we ran a couple of benchmarks that confirmed the efficiency in terms of function calls for the *odd-even* and *sieve* programs (explained in next section). In particular, Fig. 4 shows the distribution of functions calls over app calls (which perform usual typed or untyped function application, saving the current frame on the dump), the tap calls (which perform usual tail call, not saving the frame), and the tca calls (which perform cast tail calls). When the last percentages are high, it means that a significant portion

bench	tail call depth	non-tail call depth	dump len.
sieve	7942	1000	1009
odd-even	10^7	1	2

Fig. 5: Benchmark of Space Consumption (Dump Size)

of typed app calls were avoided. Another important metric is the size of the dump stack, which should be of the same order as the maximum depth of non-terminal recursive calls. This is the case both for *sieve* and for *odd-even* as shown in Figure 5. For a less empiric characterization we plan as future work to build up on [8] and find a class of space-efficiency to which our machine belongs.

5 PERFORMANCE

This work aims at alleviating the performance issues of a language with gradual set-theoretic types. The two key ingredients we developped to achieve this are:

(1) Cast compression using set-theoretic types to prevent the accumulation of multiple casts on an expression.
(2) Compressing casts in tail position using cast compression on the dump stack.

We chose our benchmarks to test the impact of these two features.

5.1 Benchmarks

We describe each benchmark, and why it was chosen to test our virtual machine.

Sieve This program finds prime numbers using the Sieve of Eratosthenes. It was among those which nailed in the coffin of sound gradual typing in Takikawa et al. [27], with a mean overhead of 100x compared to the untyped running times. The mean overhead we obtain with our solution is 7.6x.

Odd-even This is the program we gave as an example in Section 2.2: it computes whether an integer is odd or even using two mutually recursive functions. Usually, in sound gradual typing, the typed version of this program is not tail recursive, which incurs bad performances compared to the untyped version. In our machine, it is tail recursive and we obtain an overhead of 1.5x, which goes down to 1.15x with symbolic optimizations.

Cast-acc This ad-hoc (and highly unrealistic) program was written to illustrate a pitfall of our current optimization using symbolic computations: it does not memoize previously done cast computations. Here, we cast a function multiple times and then use it repetitively, and the cost is higher with symbolic computations because casts on the function have to be recomputed with each use. Knowing this pitfall exists makes the good performances of the **SymbolicCap** runtime in the other benchmarks even more interesting.

Polyadhoc This program, similarly to **Odd-even**, runs mutually recursive functions, one which is statically typed, the other which is partially typed. Except that this time, the cast inserted in the code is not a basic type, but an intersection of arrow types. It is essentially the same as **Odd-even**, except that domains and result types are harder to compute.

5.2 Experimental Setup

We implemented our machine in OCaml. The project is about 3000 lines of code (available at https://github.com/gliboc/cast-machine),

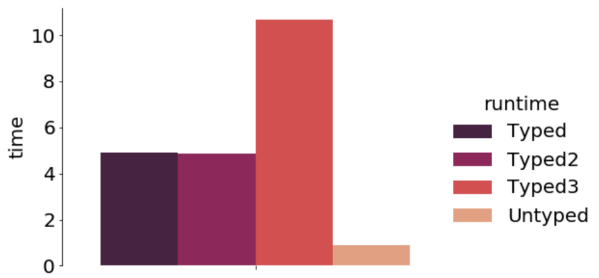

Fig. 6: Sieve Benchmark

and relies on the CDuce library for its representation of types (https://gitlab.math.univ-paris-diderot.fr/cduce/cduce). Our tests were executed on a machine with a CPU i7-5600U (Clock 2,6 GHz).

Runtimes. The execution times of the four benchmarks described above are presented in Figure 7.

- The **Typed** runtime corresponds to a typed execution of the program. There can be several typed versions, as in the case of **Sieve**, or just one (the other cases).
- The **Symbolic** runtime corresponds to using casts where the operations of domain and result are symbolic.
- The **SymbolicCap** runtime corresponds to the case in which also the intersection of type-casts are symbolic.
- The **Untyped** corresponds to running the program without any compiler-inserted casts.

What Figure 7 shows is that, as expected, the typed execution is much slower than the untyped one. However, this overhead is reasonable compared to the results obtained for sound gradual typing by Takikawa et al. [27]. Moreover, the results show that there is room for improvement, as we obtain better results by using symbolic computations for casts. Finally, the benchmarks, in particular **cast-acc**, strongly suggest that addition of memoization to the computation of casts may further improve performance, yielding important speed ups in particular cases.

5.3 Evaluation Criteria

There are many ways to optimize the code running in a virtual machine. Here, we are interested in measuring the overhead that occurs when executing different versions of a program annotated with gradual types. A gradual type system allows both fully typed and untyped versions of a program, and so does the machine we built. The difference comes from using different reduction rules: the untyped one, $[\beta_\star]$, or the typed one $[\beta_\Box]$. By comparing the execution time of these different versions of the same programs, which we call configurations, we intend to measure directly the overhead of gradual typing on our machine.

Takikawa et al. [27] define the notion of N-deliverable to compare different configurations: a configuration is N-deliverable if its performance is no worse than an Nx slowdown compared to the completely untyped configuration. Another interesting definition is that a configuration is N/M-usable if its performance is worse than an Nx slowdown and no worse than an Mx slowdown compared to the completely untyped configuration. Finally, for any choice of N and M, a configuration is unacceptable if it is neither N-deliverable nor N/M-usable. For $N = 3$ and $M = 10$, the *sieve* benchmark in [27] concludes that most configurations incur an

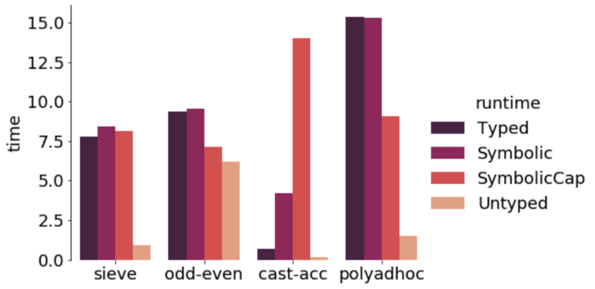

Fig. 7: Full Benchmarks

overwhelming overhead (on average, more than 100 times the untyped running time). We reproduce this benchmark on our machine for comparison, and obtain the results of Figure 6.

The four bars of Figure 6 correspond to four configurations of interest of the program *sieve*. This program consists in two modules: (1) an implementation of *streams* using thunks and pairs; (2) an implementation of the Sieve of Eratosthenes using the *streams* module.

The different configurations we consider are obtained by variating the degree of annotations in each of these modules. In particular:

- **Untyped**: no type annotations, no type-casts
- **Typed**: we annotated two functions in each module
- **Typed2**: only the module *streams* is completely annotated
- **Typed3**: only the code of *sieve* is completely annotated

We obtain an overhead of 5x for the **Typed2** configuration, meaning it is a 5-deliverable according to the metrics of Takikawa et al. [27], and therefore a 3/10-usable. Similarly, **Typed3** has an overhead of around 10x compared to the untyped *sieve*. This is still significant, but it is way faster than the maximum overhead of more than 110x obtained in Takikawa et al. [27] and still acceptable by the reasonable standards of 3/10-usability.

5.4 Impact of Set-Theoretic Operators

Our implementation relies on the CDuce implementation to compute four type operators: (1) decide subtyping $\leq$; (2) the domain operator dom; (3) the apply operator $\circ$; (4) the intersection of types. We tried to minimize the use of these operations in our machine by using a symbolic representation of casts. Our implementation of this technique yields the results seen in Figure 7. We notice in particular that the most interesting configuration is the one which uses symbolic intersections.

6 RELATED WORK

We followed an approach similar to the one from Siek and Garcia [22], who built abstract machines for the gradually-typed lambda calculus, and used parameterization to model several different semantics for gradual typing such as eager or lazy cast checking, as well as different kinds of blame tracking. This broad approach of experimenting the different features of gradual typing with abstract machine before plugging gradual type systems into real systems reflects the current evolving state of gradual typing.

The problem of cast accumulation was recognized by Herman et al. [16], who proposes a solution based on the coercions of Henglein [15]. Siek et al. [26] present an efficient algorithm for compressing coercions. Alternatively, Siek and Wadler [25] propose a solution

based on compressing a sequence of casts into a pair of casts, to and from the least upper bound of the types (with respect to type precision). One can view the present paper as generalizing this approach to languages with set-theoretic types and subtyping.
Contracts are a generalization of casts to handle arbitrary predicates [12]. Greenberg [14] proves that the space overhead for contracts can be bounded by a constant by taking care to never wrap the same contract on a value multiple times. Feltey et al. [11] implement and evaluate this approach in the Racket contract library, which underlies the implementation of sound gradual typing in Typed Racket. The approach that we present, based on the BDD representation of types, enables finer-grained sharing which we conjecture leads to better compression.
While the present paper focuses on efficiency but leaves out blame tracking, Keil and Thiemann [17] develop an operational semantics for intersection and union types that includes blame tracking, but they do not consider space efficiency.
On a more set-theoretic perspective, an interesting approach is dedicated to the inference of interfaces — the type constraints of functions — and in particular of intersections of interfaces. This type inference makes it possible to fully annotate a module more quickly, in order to faster bridge the gap between untyped and partially-typed performances. The paper [6] treats the subject, but in a context of non-gradual set-theoretic types.

7 FUTURE WORK

Blame tracking. In gradual typing, blame tracking makes it possible to find which cast in the code led to a failure. It should satisfy two properties: *blame safety*, and *type safety*. Blame safety means that an expression that could reduce to a value should never be blamed, and soundness that a well-typed expression can either reduce to a value, diverge, or be blamed. Our machine has difficulties in assigning blame, because the compression of casts using type intersections lose the information of blame labels.
However, it might be possible to compress the labels as well as the casts. We conjecture that if a function terminates, then the sequence of blame labels of its casts can be expressed as a regular expression whose size is bounded according to the number of blame labels in the original program. This regular expression would record the arrival of each casts on an expression: therefore, when a type-cast fails on a value, it would be possible to blame the earliest type-cast that was incompatible with the value. A fallback solution would be to handle sets of blame labels, of which there finitely many, instead of regular expressions, but this would yield far less precise blames.

Benchmarks and Language Extensions. We would also like to test more thoroughly our implementation by adapting the rest of the benchmarks of Takikawa et al. [27], and by finding other tests specific to set-theoretic types. There are also some language features that could be of interest once our machine is sufficiently improved, such as (i) using type intersections in annotations—currently it is only possible to annotate functions with a function type and (ii) extending the type system with polymorphism.

8 CONCLUSION

The goal of this work was to study the implementation of functional languages using a gradual type system with set-theoretic types. Our main contribution is our technique of combining intersection types and domain caching to obtain a space efficient compression of cast compositions. This, combined with various other implementation techniques we described (caching of casts in closures, use of the dump to efficiently implement cast application in tail position, the symbolic computation of type operations) yields an implementation satisfactory in space consumption and is not extremely penalizing in time consumption. The time overhead due to set-theoretic types is still too important but, as we discussed in Section 5 so is the room for improvement, that we plan to explore in future work

REFERENCES

[1] S. Bauman, C. F. Bolz-Tereick, J. G. Siek, and S. Tobin-Hochstadt. Sound gradual typing: Only mostly dead. *PACMPL*, 1(OOPSLA):54, 2017.
[2] G. Bierman, M. Abadi, and M. Torgersen. Understanding TypeScript. In *European Conference on Object-Oriented Programming*, pages 257–281. Springer, 2014.
[3] A. Bonnaire-Sergeant, R. Davies, and S. Tobin-Hochstadt. Practical optional types for Clojure. In *ESOP 2016*, pages 68–94. Springer, 2016.
[4] G. Castagna. Covariance and controvariance: a fresh look at an old issue (a primer in advanced type systems for learning functional programmers). *Logical Methods in Computer Science*, 2019. https://arxiv.org/abs/1809.01427. To appear.
[5] G. Castagna and V. Lanvin. Gradual typing with union and intersection types. *PACMPL*, 1(ICFP):41, 2017.
[6] G. Castagna, K. Nguyen, Z. Xu, and P. Abate. Polymorphic functions with set-theoretic types: Part 2: Local type inference and type reconstruction. In *POPL '15*, volume 50, pages 289–302, 2015.
[7] G. Castagna, V. Lanvin, T. Petrucciani, and J. G. Siek. Gradual typing: a new perspective. *PACMPL*, 3(POPL):16, 2019.
[8] W. Clinger. Proper tail recursion and space efficiency. In *PLDI '98*, 1998.
[9] Facebook. Flow documentation, . https://flow.org/en/docs/lang/.
[10] Facebook. Hack documentation, . https://docs.hhvm.com/hack/.
[11] D. Feltey, B. Greenman, C. Scholliers, R.B. Findler, and V. St-Amour. Collapsible contracts: Fixing a pathology of gradual typing. *PACMPL*, 2(OOPSLA):133:1–133:27, October 2018.
[12] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP '02*, pages 48–59, October 2002.
[13] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM (JACM)*, 55(4):19, 2008.
[14] M. Greenberg. Space-efficient manifest contracts. In *POPL '15*, pages 181–194. ACM, 2015.
[15] Fritz Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, June 1994.
[16] David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. *Higher-Order and Symbolic Computation*, 23(2):167, 2010.
[17] M. Keil and P. Thiemann. Blame assignment for higher-order contracts with intersection and union. In *ICFP 2015*, pages 375–386, 2015.
[18] A. Kuhlenschmidt, D. Almahallawi, and J. G. Siek. Toward efficient gradual typing for structural types via coercions. In *PLDI '19*, 2019.
[19] J. Lehtosalo and D. J. Greaves. Language with a pluggable type system and optional runtime monitoring of type errors. In *Workshop on Scripts to Programs (STOP)*, 2011.
[20] A. Rastogi, N. Swamy, C. Fournet, G. Bierman, and P. Vekris. Safe & efficient gradual typing for TypeScript. In *POPL '15*, pages 167–180. ACM, 2015.
[21] G. Richards, F. Zappa Nardelli, and J. Vitek. Concrete types for TypeScript. In *ECOOP 2015*, 2015.
[22] J. G. Siek and R. Garcia. Interpretations of the gradually-typed lambda calculus. In *Workshop on Scheme and Functional Programming*, pages 68–80, 2012.
[23] J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.
[24] J. G. Siek and M. Vachharajani. Gradual typing with unification-based inference. In *Symposium on Dynamic languages*, page 7. ACM, 2008.
[25] J. G. Siek and P. Wadler. Threesomes, with and without blame. In *POPL 2010*, pages 365–376, 2010.
[26] J. G. Siek, P. Thiemann, and P. Wadler. Blame and coercion: together again for the first time. In *PLDI '15*, pages 425–435, 2015.
[27] A. Takikawa, D. Feltey, B. Greenman, M. S. New, J. Vitek, and M. Felleisen. Is sound gradual typing dead? In *POPL '16*, pages 456–468, 2016.
[28] S. Tobin-Hochstadt and M. Felleisen. Interlanguage migration: from scripts to programs. In *DSL '06*, pages 964–974. ACM, 2006.
[29] Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of Typed Scheme. In *POPL '08*, pages 395–406. ACM, 2008.
[30] M. M. Vitousek, A. M. Kent, J. G. Siek, and J. Baker. Design and evaluation of gradual typing for Python. In *DSL '14*, pages 45–56, 2014.