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

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 a string inside of statically typed code!)

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 (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 perressive 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 masquerade 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 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 \( \text{Int} \to ? \) 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 ? \to \text{Int} to foo, ? \to ? to bar, and \text{Int} \to ? to baz. This is so because in gradual typing the types \text{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, \text{Int}). 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 \text{Int} is expected (e.g., the x in the body of foo) and use an expression of a static type such as \text{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 \text{e + 1}, the gradual type checker will consider that ? is consistent with \text{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(Int) + 1 where e(Int) is a type-cast expression that dynamically checks whether the result of e has type \text{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 \text{Bool} \to ? \to ? 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>)

But looking again at this example, we see that its derived type \text{Bool} \to ? \to ? 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 \text{t}_1 \land \text{t}_2, union types \text{t}_1 \lor \text{t}_2, and negation types \neg \text{t}. These constructors respect naive set intuitions, so that an expression of type \text{t}_1 and \text{t}_2 can be given the type \text{t}_1 \land \text{t}_2, etc. In our previous example, they allow the programmer to annotate the argument x more precisely:

let f (condition) (x : (\text{Int} \lor \text{Bool}) \land ?) =
  if condition then x<Int> + 1 else ¬(x<Bool>)

where "\land" denotes union and "\&" denotes intersection. The union \text{Int} \lor \text{Bool} indicates that a value of this type is either an integer or a Boolean, and the intersection indicates that x has both type \text{Int} \lor \text{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. [6] is able to deduce for this code example the type:

\text{Bool} \to ((\text{Int} \lor \text{Bool}) \land ?) \to (\text{Int} \lor \text{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 \text{Int} \lor \text{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, \text{Int} \times \text{Int} \to \text{Int} and \text{Bool} \to \text{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 \text{r} = (\text{Int} \to \text{Int}) \land (\text{Bool} \to \text{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 the functions of this type), denoted by \text{dom} \text{r} \to \text{Int} \lor \text{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 \text{Int} \lor \text{Bool}. Therefore, an operator on types needs to be defined which we denote by \circ. More precisely, we denote by \text{t}_1 \circ \text{t}_2 the type of an application of a function of type \text{t}_1 to an argument of type \text{t}_2. In the example with \text{r} = (\text{Int} \to \text{Int}) \land (\text{Bool} \to \text{Bool}), it gives \text{r} \circ \text{Int} = \text{Int} \lor \text{Bool} = \text{Bool}, and \text{r} \circ (\text{Int} \lor \text{Bool}) = \text{Int} \lor \text{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 \text{r} 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 byte-code. 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. [6] tweaked for efficient implementation. In particular, we modify the syntax of casts to remove unessential parts and to minimize 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

$$\langle \tau_1 \rightarrow \tau_2 \rangle$$

where $$\tau_2$$ is used to store information about the domain of $$\tau_1$$ when this is a function type and $$\emptyset$$ otherwise. We do not explain how to compile a gradually-typed program into a program of the cast language since this is essentially the same as in [6]. 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 $$\mathcal{B}$$ of basic types (e.g., $$\mathcal{B} = \{\text{Int, Bool}\}$$). Following the approach of Castagna et al. [6], we define both gradual and static (a.k.a. non-gradual) set-theoretic types as the terms produced coinductively by the following grammars:

<table>
<thead>
<tr>
<th>Type</th>
<th>Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>$$\tau ::= b \mid t \times t \mid t \rightarrow t \mid \lnot t \mid \emptyset$$</td>
</tr>
<tr>
<td>Gradual</td>
<td>$$\tau ::= \emptyset \mid b \mid t \times \tau \mid t \rightarrow \tau \mid \lnot t \rightarrow \tau \mid \lnot t \mid \emptyset$$</td>
</tr>
</tbody>
</table>

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 [6] for more explanation about these restrictions. We introduce the following abbreviations for gradual types:

- $$\tau_1 \land \tau_2 \equiv \lnot (\lnot \tau_1 \lor \lnot \tau_2)$$,
- $$\tau_1 \lor \tau_2 \equiv \tau_1 \land \lnot \tau_2$$,
- $$\top \equiv \lnot \bot$$,
- and likewise for static types. We refer to $$\times$$ and $$\rightarrow$$ as type constructors and to $$\land$$, $$\lor$$, $$\lnot$$, and $$\top$$ 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 $$\emptyset$$ and $$\top$$ 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. [6]. 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:

$$\begin{align*}
\tau \leq \tau' & \quad \tau' \leq \tau \\
\tau \leq \tau & \quad \tau \leq \tau \\
\lnot \tau \leq \lnot \tau & \quad \lnot \tau \leq \lnot \tau \\
\tau_1 \land \tau_2 \leq \tau_3 & \quad \tau_1 \land \tau_2 \leq \tau_3 \\
\tau_1 \lor \tau_2 \leq \tau_3 & \quad \tau_1 \lor \tau_2 \leq \tau_3 \\
\tau_1 \rightarrow \tau_2 \leq \tau_3 & \quad \tau_1 \rightarrow \tau_2 \leq \tau_3 \\
\tau_1 \times \tau_2 \leq \tau_3 & \quad \tau_1 \times \tau_2 \leq \tau_3 \\
\top \leq \tau \quad & \quad \bot \leq \tau \\
\lnot \top \leq \lnot \tau & \quad \lnot \bot \leq \lnot \tau \\
\top \leq \bot & \quad \bot \leq \top
\end{align*}$$

In particular, $$\bot$$ is only a subtype of itself and of $$\top$$, and the same holds for $$\top$$.

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:

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

This function can be given the dynamic type $$\text{Int} \rightarrow ?$$. Since, intuitively, $$\bot$$ stands for any type (or the absence of type), this function can be cast to type $$\text{Int} \rightarrow \text{Int}$$. 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 $$\text{Int} \rightarrow \text{Int}$$ is safe and will not produce a runtime error. However, it is also clear that this function cannot be cast to type $$\text{Int} \rightarrow \text{Int}$$ 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 $$f : (\text{Int} \rightarrow \text{Int})$$ 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 $$e(\tau)$$ denotes a cast of the expression $$e$$ to the type $$\tau$$ and $$v$$ denote values:

$$\beta[\beta_0] : \langle \tau_1 \rightarrow \tau_2 \rangle \langle v_1 \rangle \langle v_2 \rangle \quad \rightarrow \quad \langle v_1 \cdot v_2(\tau_1)(\tau_2) \rangle$$

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 [7], the rule becomes:

$$\beta[\beta_0] : \langle \tau_1 \rangle \langle \tau_2 \rangle \quad \rightarrow \quad \langle v_1 \cdot v_2(\text{dom}(\tau))(\tau \circ \text{type}(\tau_2)) \rangle$$

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

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:

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

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 $$\text{Int} \rightarrow ?$$,
drives the type-checker to add explicit type-casts that enforce \( ? \) to be the return type of \( \text{odd} \). In the same way, in the body of \( \text{even} \), the call to \( \text{odd} \) is considered to return an element of type \( ? \) and has to be cast to \( \text{Bool} \). Now, a naive evaluation of \( \text{odd} 5 \) would yield:

\[
\begin{align*}
\text{odd} 5 & \rightarrow (\text{even} \ 4)<? > \\
& \rightarrow (\text{odd} \ 3)<\text{Bool}><?> \ \\
& \rightarrow (\text{even} \ 2)<?><\text{Bool}><?> \\
& \rightarrow (\text{odd} \ 1)<\text{Bool}><?><\text{Bool}><?> \ \\
& \rightarrow (\text{even} \ 0)<?><\text{Bool}><?><\text{Bool}><?>
\end{align*}
\]

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 Graal 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 \( E(c_1)(c_2) \) and computes \( E(c_1)c_2 \) where the \( \Join \) 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: an expression casts than on the form of their types). This operation is straightforward thanks to set-theoretic types: an expression \( E(\tau)(\tau') \) is “compressed” to \( E(\tau \land \tau') \). 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 \( v((\text{Int} \rightarrow \text{Int})(? \rightarrow ?)(\text{Bool} \rightarrow \text{Bool}))' \) must never succeed, since this would result in the argument \( v' \) being cast from \( \text{Bool} \) to \( ? \) Int before being passed to the function \( v \), and no value can be of both types \( \text{Bool} \) and \( \text{Int} \). However, if we were to merge these casts naively, we would obtain the application \( v((\text{Int} \rightarrow \text{Int}) \land (? \rightarrow ?) \land (\text{Bool} \rightarrow \text{Bool}))(v') \). Since \( (\text{Int} \rightarrow \text{Int}) \land (? \rightarrow ?) \land (\text{Bool} \rightarrow \text{Bool}) \) is a subtype of \( (\text{Int} \lor \text{Bool}) \lor (\text{Int} \lor \text{Bool}) \lor (\text{Int} \lor \text{Int}) \), then in particular this application will succeed whenever \( v' \) is of type either \( \text{Int} \lor \text{Bool} \), 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 \( E(\tau)(\tau) \land (\tau)(\tau) \land (\tau)(\tau) \) into \( E(\tau \land \tau \land \tau \land \tau) \). Adding this information to casts, the previous application becomes \( v((\text{Int} \rightarrow \text{Int}) \land (? \rightarrow ?) \land (\text{Bool} \rightarrow \text{Bool}))'(v') \), which reduces to \( v((\text{Int} \rightarrow \text{Int}) \land (? \rightarrow ?) \land (\text{Bool} \rightarrow \text{Bool}))\text{Int} \land \text{Bool} \lor \text{Bool} \lor \text{Bool} \lor \text{Bool} \). This expression will then always fail, since the type of the argument is now checked against the exponent, and \( \text{Int} \land \text{Int} \lor \text{Bool} \) is empty (since \( \text{Int} \lor \text{Bool} \) is so) and thus no value can have this type.

### 2.4 Big step semantics

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:

\[
\begin{align*}
\text{Func} \quad u & := o \mid \mu \tau \rightarrow t \ f \ x \ E \ \\
\text{Val} \quad v & := k \mid (v, v) \mid [u, E, (\tau)']^m \ \\
\text{Mark} \quad m & := \star \mid \square
\end{align*}
\]

The operator type is defined on values \( v \in \text{Val} \) as:

\[
\begin{align*}
\text{type}(u, E, (\tau_1)^m) & = \tau_1 \\
\text{type} k & = E(k) \land ? \land ? \\
\text{type} (v_1, v_2) & = v_1 \times \text{type} v_2
\end{align*}
\]

where \( \mathcal{B} \) maps non functional constants into their basic type. Our values include closures \([u, E, (\tau)']^m \), which are formed by a functional expression \( u \), an environment \( E \) which is a finite function \( \mathcal{V} \rightarrow \mathcal{V} \), 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 \( (\tau)’ \) is waiting to be applied to the function inside the closure; when \( m = \star \), \( (\tau)’ \) 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 \( \beta_i \)) while \( \star \)-closures are usual functions without any cast (they will be reduced by the usual \( \beta \)). 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., \( +, \text{inc}r, \ldots \)), and because casts behave differently on these two kinds of constants.

---

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.
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.

$$E \vdash \mu f . \tau \to \tau' \; f \; x \; E \Rightarrow [\mu f . \tau \to \tau' \; f \; x \; E, \tau_1 \to \tau_2]^{\star}$$

Then, the following rules handle function application. The first, $[\beta_\star]$, is a standard untyped function application that corresponds to the standard $\beta$-reduction. The second, $[\beta_2]$, enforces cast constraints on the arguments and on the result of the application, which this time corresponds to the rule $[\beta_1]$ outlined earlier. We denote by $E[\{x := v\}]$ a copy of $E$ which outputs $v$ when given $x$.

$$\begin{array}{c}
E \vdash E_1 \Rightarrow v_1 = [\mu f . \tau \to \tau' \; f \; x \; E, E', \tau]^{\star} \\
[\beta_\star] \\
E \vdash E_2 \Rightarrow \forall x : v_2 \quad E'[\{x := v_2 \; f := v_1\}] \Rightarrow \forall \quad \Rightarrow E \Rightarrow \forall
\end{array}$$

$$\begin{array}{c}
E \vdash E_1 \Rightarrow [u, E', (\tau_1 \to \tau_2)]^{\top} \\
E \vdash E_2 \Rightarrow \forall x : v_2 \quad E'[\{x := v_2 \; f := v_1\}] \Rightarrow \forall
\end{array}$$

$$\begin{array}{c}
[\beta_2] \\
E \vdash (v_1 \; v_2) (\tau_1 \circ \text{type}(\tau_2)) \Rightarrow \forall
\end{array}$$

Let us see how $[\beta_2]$ works. When applying a closure $[u, E', (\tau_1 \to \tau_2)]^{\top}$, the index type $\tau_2$ should already capture the type of the argument. Therefore, the argument $E_2$ is evaluated to a value $v_2$ and then cast to $v_2$ using $\tau_2$. During that time, the type $\tau = \tau_1 \circ \text{type}(v_2)$ is computed in order to best approximate the result type of the function call. Finally, $\tau_1 \; v_2$ is 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:

$$\begin{array}{c}
E \vdash E \Rightarrow [u, E', (\tau_1 \to \tau_2)]^{m} \\
E \vdash E(\tau_1)^{\tau_2} \Rightarrow [u, E', (\tau_1 \land \tau_2 \land \tau_1)]^{\top}
\end{array}$$

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 $\top$ in order to signify that it should be applied using rule $[\beta_2]$, 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 $42<?\text{Int}_\top\land\rightarrow?$ or $42<\text{Bool}\land\rightarrow?$, the only parts of these casts that have a consequence are $\text{Int}$ or $\text{Bool}$, since $?$ could represent anything. Therefore, we can erase $?$ in both these casts, and re-apply the newly-obtained cast. To this purpose we define the extremum of gradual types in order to be able to cast constants to gradual types.

**Definition 1. (Gradual Extrema)** For every gradual type $\tau$,

- the gradual maximum $\tau^{\top}$ is obtained by replacing every contravariant occurrence of $?$ by $\top$, and every covariant occurrence by $\bot$

- the gradual minimum $\tau^{\bot}$ is obtained by replacing every contravariant occurrence of $\bot$ by $\top$, and every covariant occurrence by $\bot$

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

$$\begin{array}{c}
E \vdash E \Rightarrow k \quad B(k) \leq \tau_1^{\bot} \\
E \vdash E \Rightarrow k \quad B(k) \not\leq \tau_1^{\bot}
\end{array}$$

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 use most of the standard rules of a simply-typed lambda-calculus with pairs plus classic subsumption for subtyping:

$$\begin{array}{c}
\text{Subsumes} \\
\Gamma \vdash E : \tau' \\
\Gamma \vdash E : \tau \\
\tau' \leq \tau
\end{array}$$

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

$$\begin{array}{c}
\text{Constant} \\
\Gamma \vdash k : B(k) \land ? \land \neg ? \\
\Gamma \vdash E : \tau
\end{array}$$

The rule for casts states that, since $\tau_1$ is the most precise type that can be given to $E(\tau_1)^{\tau_2}$ if the cast succeeds, we simply give this type to the expression, provided $E$ is well-typed. Notice that both $\tau$ and $\tau_2$ 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 $E(k)$ (which is its expected type), but can also be implicitly downcast to type $?$ and to $\neg ?$. 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 $42<\text{Int}^{\top}$: 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 semantics of the program. However, this means in particular that $42<\text{Int}^{\top}$, which is of type $?$, reduces to $42$, which would intuitively be of type $\text{Int}$. Such a reduction would violate type preservation, hence the need for the rule $[\text{Const}]$ which allows constant to be implicitly consider of type $\tau$.

The part $\neg ?$ can be understood with the same reasoning, as $? \land \neg ?$ are actually semantically equivalent (but incomparable for subtyping). Indeed, since $?$ can intuitively represent any type $\tau$, it can also represent the negation of any type $\neg \tau$. Therefore, $\neg ?$ can represent any double negation $\neg \neg \tau$, thus any type $\tau$.

Using this type system, we can then prove the traditional lemmas.

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

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

From which soundness follows.

Theorem 1. If $\Gamma \vdash E : \tau$ and $E$ is closed, then either $E$ diverges, or $E \rightarrow^* \nu$ for some value $\nu$, or $E \rightarrow^* \text{Fail}$.

In order to prove that using $[\beta_2]$ 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).

$$
\begin{align*}
\text{c} &::= \emptyset \mid \text{instr} : c \\
\text{instr} &::= | \text{push obj} \quad \text{Stack push} \\
&| \text{app} \mid \text{ret} \\
&| \text{tap} \\
&| \text{tca} \kappa \\
&| \text{cast} \\
&| \text{ifz} (c, c) \\
&| \text{let} \mid \text{end} \\
&| \text{pair} \\
\text{obj} &::= n | k \mid o \mid (c, \kappa) \mid \kappa \\
\text{a} &::= \times | + | - | = | \text{fst} | \text{snd}
\end{align*}
$$

The push instruction injects identifiers, constants, closures, and types into the execution by pushing them onto the stack. Pair of types $\kappa$ 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 $\rho$ and calls $\text{lookup } \rho x$ which returns the index of the first occurrence of $x$ in $\rho$.

The compilation process is decomposed into two functions. The function $C[\cdot]_\rho : \text{Expr} \rightarrow \text{Bytes}$ is the general one. It calls $T[\cdot]_\rho$ 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:

$$
\begin{align*}
C[c]_\rho &= \text{push } c \\
C[x]_\rho &= \text{push } (\text{lookup } \rho x) \\
C[\mu^{\tau_1 \rightarrow \tau_2} f \times. E]_\rho &= \text{push } (T[E]_{f \times. x. \rho}^{\tau_1 \rightarrow \tau_2}) \\
C[E(\tau_1)\_\rho] &= \text{push } \langle \tau_1 \rangle^{\tau_1} \\
C[E_1 E_2]_\rho &= C[E_1]_\rho; C[E_2]_\rho \\
C[\text{let } x = E_1 \text{ in } E_2]_\rho &= C[E_1]_\rho; C[E_2]_x \rho; \text{let}; C[E_2]_\rho; \text{end} \\
C[\text{ifz} (c_1, c_2)]_\rho &= C[c_1]_\rho; C[c_2]_\rho; \text{tap} \\
T[E_1 E_2]_\rho &= C[E_1]_\rho; C[E_2]_\rho; \text{tap} \\
T[\text{let } x = E_1 \text{ in } E_2]_\rho &= C[E_1]_\rho; C[E_2]_x \rho; \text{let}; T[E_2]_\rho \\
T[E]_\rho &= C[E]_\rho; \text{ret}
\end{align*}
$$

Our machine follows a classic eval-apply pattern with some specificities. In particular, the evaluation of a $\lambda$-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: according to the case, this instruction compiles 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 push φs(obj, s) function handles adding elements to the stack.

- push φs(n, s) \rightarrow e(n). s
- push φs(k, s) \rightarrow k. s
- push φs(c, k, s) \rightarrow [c, e, k] s
- push φs(k, s) \rightarrow k. s

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

- reflect τ ⊕ τ = (τ ∧ τ)⊥
- reflect τ ⊕ τ = (τ ∧ τ)⊥τ

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

- cast(k, τ1 ⊕) = k if \( k \in \tau \)
- cast([c, e, \langle \tau \rangle m], τ1 ⊕) = [c, e, \langle \tau \rangle \tau ∩ τ1 ⊕] if \( ∩ ≠ \bot \)
- cast(β, k) = \( \Omega \) in all other cases

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 \( β_\tau \), tail function calls \( \{tβ_\tau\} \), and cast tail function calls \( \{cβ_\tau\} \). 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: \( β_\tau \), \( tβ_\tau \), and \( cβ_\tau \) 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 \( \{R_\tau\} \), which replaces the current frame by the top frame on the dump; and a typed one \( \{R_2\} \), 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 2.3. By inlining the function even into the definition of odd, we can compile the odd into a piece of bytecode codd, which is put inside the closure for odd, \( \{codd, e\}', \langle \text{Int} → \text{Int}\rangle s' \), and we have

\[ C[\text{odd 5}] = \text{push 5; push odd; app} \]

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

- The first call to odd 5 is handled by regular application:

  \[ (\text{app}, e, \{codd, e\}', \langle \text{Int} → \text{Int} \rangle s') \rightarrow (codd, (n := 5), e', \varnothing, (\varnothing, e)) \]

  - The code of odd executes and, since \( n = 5 \) is not zero, it runs the code for \( (\text{even } (n-1)) \)? . This expression is in tail position and consists of a cast on a function call. Therefore, the \( \{cβ_\tau\} \) 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.

  \[ \{\text{tca ?}, \langle n := 5 \rangle, e', \{c\text{even-}e''\}, \langle \text{Int} \rightarrow \text{Int} \rangle s' \} \rightarrow (\text{tap}, \langle n := 5 \rangle, e', \{c\text{even-}e''\}, \langle \text{Int} \rightarrow \text{Int} \rangle s' \} \rightarrow (\varnothing, e') \]

  - The execution of \( c\text{even} \) is similar to the one of \( c\text{even} \) and adds to the dump the cast \( \text{Bool} \) which composes with the cast \( \varnothing \) already on the top of the dump, yielding

    \[ \rightarrow (\text{codd}, \langle n := 3 \rangle, e', \varnothing, (\text{Bool} \land ?), (\varnothing, e)) \]

    - These cast calls keep decreasing the value of \( n \), building up a single type-cast on top of the dump, until \( \text{even } 0 \) 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:

    \[ \{\text{push true}; \text{ret}, \langle n := 0 \rangle, e'', \varnothing, (\text{Bool} \land ?), (\varnothing, e) \rightarrow (\text{ret}, \langle n := 0 \rangle, e'', \text{true, } (\text{Bool} \land ?), (\varnothing, e) \rightarrow (\text{cast}; \text{ret}, \langle n := 0 \rangle, e'', (\text{Bool} \land ?), \text{true}, (\varnothing, e) \rightarrow (\text{ret}, \langle n := 0 \rangle, e'', \text{true, } (\varnothing, e) \rightarrow (\varnothing, \text{e, true, } (\varnothing, e)) \]

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 \( \nu \) into the corresponding machine values \( \nu \), and extend it to environments \( [\_ \_ \_ ] \), which is similar to the one of \( [\_ \_ \_ ] \), and adds to the dump the cast \( \text{Bool} \) which composes with the cast \( \varnothing \) already on the top of the dump, yielding

\[ \rightarrow (\text{codd}, \langle n := 3 \rangle, e', \varnothing, (\text{Bool} \land ?), (\varnothing, e)) \]

These cast calls keep decreasing the value of \( n \), building up a single type-cast on top of the dump, until \( \text{even } 0 \) 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:

\[ \{\text{push true}; \text{ret}, \langle n := 0 \rangle, e'', \varnothing, (\text{Bool} \land ?), (\varnothing, e) \rightarrow (\text{ret}, \langle n := 0 \rangle, e'', \text{true, } (\text{Bool} \land ?), (\varnothing, e) \rightarrow (\text{cast}; \text{ret}, \langle n := 0 \rangle, e'', (\text{Bool} \land ?), \text{true}, (\varnothing, e) \rightarrow (\text{ret}, \langle n := 0 \rangle, e'', \text{true, } (\varnothing, e) \rightarrow (\varnothing, \text{e, true, } (\varnothing, e)) \]

3.5 Symbolic casts

A drawback of this virtual machine is that it relies on high-order operations on types: \( \text{dom, } \varnothing \land \) see (the three \( \beta_\tau \) 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 concentrate on \( \text{dom} \) and \( \varnothing \) 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 \( \kappa \) by symbolic casts \( \Sigma \):

\[ \Sigma \text{Bytes} \] \[ c := \varnothing | \text{instr}; c \]
\[ \Sigma \] \[ \kappa ] \[ \Sigma \Sigma ) \]
\[ \text{instr} := \_ \_ \_ \text{tca } \_ \_ \_ \]

Symbolic casts encode in their structure the calls to domain \( \text{dom} \) and cast composition \( \varnothing \), and they are lazily evaluated by \( \text{eval} \):
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 type casts, and the type operators (dom, Ω) 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 → * E’, then size(E’) ≤ c · size(|E'|).

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

$$\text{eval}(\text{dom } \Sigma) = (\tau_2)\text{dom } \tau_2 \quad \text{with } \text{eval}(\Sigma) = \tau_1 \tau_2$$

$$\text{eval}(\text{app }, \Sigma) = (\tau_r)\text{dom } \tau_r \quad \text{with } \text{eval}(\Sigma) = \tau_1 \tau_2$$

Structures. The structures of the machine are slightly different as well, as again we replace pairs of types \(\hat{\kappa}\) with symbolic casts \(\Sigma\).

$$\Sigma \text{Val} \quad \nu \quad ::= \quad c \mid (\nu, \Sigma) \mid [c, \Sigma, \nu] \quad \text{Machine values}$$

$$\Sigma \text{Val} \quad s \quad ::= \quad \emptyset \mid \nu \cdot s \mid \Sigma \cdot s \mid \Omega \cdot s \quad \text{Operand stack}$$

$$\Sigma \text{Val} \quad d \quad ::= \quad \emptyset \mid \Sigma \cdot d \mid (c, e), d \quad \text{Control stack}$$

Parameter functions. Since intersections are not symbolic operators, the composition of two symbolic pairs \(\Sigma_1, \Sigma_2\) is defined as:

$$\tau_1 \wedge \tau_2 = \tau_1 \tau_2$$

$$\text{eval}(\Sigma_1) \wedge \text{eval}(\Sigma_2)$$

The transition rules of this virtual machine with symbolic casts stay the same, except for the three rules described in Figure 2.
Fig. 2: Modified transitions of the virtual machine with symbolic casts (with $\Sigma_r = \text{app}_{\text{type}}(x)$ and $\Sigma_d = \text{dom}(x)$)

$$\begin{align*}
\text{size } x &= 1 \\
\text{size } c &= u(c) \\
\text{size } (E_1 E_2) &= \text{size } (E_1, E_2) = 1 + \text{size } E_1 + \text{size } E_2 \\
\text{...}
\end{align*}$$

where $u(c)$ denotes the size of the representation of $c$ (e.g., $u(n) = 64$ bits for an integer $n$). We see that the difference between the sizes of $E$ and of $|E|$ is given by the size of the types used in call.

By 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 the key to 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.

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 Binary Decision Diagrams (BDD). 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 $B$ of basic types. Gradual set-theoretic types are the possibly infinite terms produced coinductively by

\[
\begin{align*}
\text{Atoms} & \quad a ::= b \mid ? \mid \tau \rightarrow \tau \\
\text{Types} & \quad \tau ::= a \mid \tau \vee \tau \mid \neg \tau \mid 0
\end{align*}
\]

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:

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

(1)

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

$$
\begin{align*}
\perp &= 1 \\
T &= 0 \\
a?B_1 : B_2 &= (a \land [B_1]) \lor (\neg a \land [B_2])
\end{align*}
$$

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 which imposes that on every path the order of the labels strictly increases. Hence the depth of a BDD is upper-bounded by the number of atoms available to build it. This yields the result:

\begin{align*}
\text{size}(B) &\leq 2^{|\alpha|} \log_2 \alpha + \sum_{a \in \mathcal{A}} \text{size } a \quad \text{def } C_{\mathcal{A}}
\end{align*}

(2)

In this formula, we consider the maximum number of nodes of a tree of depth $\alpha$, which is $2^\alpha$. All the distinct atoms are stored separately using size $\Sigma_{a \in \mathcal{B}} \text{size } a$, each node of the tree being a reference to a stored atom (i.e. of size log_2 $\alpha$). 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 $E \rightarrow E'$, then every type-cast occurring in $E'$ is represented by a BDD using exclusively atoms from $\mathcal{A}$.

Corollary 5. Any type-cast $(\tau_1)^{\tau_2}$ in $E'$ is bounded in size, by size $(\tau_1)^{\tau_2} \leq 2C_{\mathcal{A}}$

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 $\land$, and the domain and result operators dom, $o_n$. Let us describe how each operation handles atoms:

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

$$
\perp \land B = B \land \perp = \perp \\
T \land B = B \land T = B
$$

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

This definition makes it clear that the atoms of $B_1 \land B_2$ 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.
We studied the third on examples and by comparisons with other......

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 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 developed 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 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 2.1: 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 fails 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), 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.

<table>
<thead>
<tr>
<th>bench</th>
<th>fun. calls</th>
<th>app (%)</th>
<th>tap(%)</th>
<th>tca (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>sieve</td>
<td>2.6 \cdot 10^9</td>
<td>59.6%</td>
<td>20.8%</td>
<td>19.6%</td>
</tr>
<tr>
<td>odd-even</td>
<td>10^7</td>
<td>~0%</td>
<td>~0%</td>
<td>~99%</td>
</tr>
</tbody>
</table>

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

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

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

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

Since union and intersections do not create new atoms, then it is enough to include in the initial set of atoms \( A \) all the atoms used in the domain \( \sigma_p \) of every single function atom \( a = \sigma_p \rightarrow \tau_p \) in \( E \).

Result operator: for \( \tau \) the function type defined above, the codomain is defined as

\[
\text{cod} \tau = \bigvee_{i \in I} \bigwedge_{p \in P_i} \tau_p
\]

We know that, for any type \( \sigma \), the result types \( \tau \circ \sigma \) will be formed using the atoms \( \tau_p \) of \( \text{cod} \tau \) — 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 \( \tau_p \) of every single function atom \( a = \sigma_p \rightarrow \tau_p \) 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.
What Figure 7 shows is that, as expected, the typed execution is which we call configurations, we intend to measure directly 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 $[\beta_{\tau}]$. 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 $N x$ 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 $N x$ slowdown and no worse than an $M x$ 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 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 varying 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
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 [5] 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 subject-reduction. Blame safety means that an expression that could reduce to a value should never be blamed, and subject-reduction 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 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
A space-efficient call-by-value virtual machine for gradual set-theoretic types

A SOURCE LANGUAGE

The syntax of the source language corresponds to the following lambda calculus with pairs, conditionals, and type annotated recursive abstractions.

\[
\text{SourceExpr} \; e \ ::= \; x \mid c \mid \mu(f: \tau \rightarrow \tau) \cdot e \mid e \; e \mid \text{let } \; x = e \; \text{ in } \; e \mid \text{ if } \; e \; e \; e \mid (e, e) \mid \pi_i e
\]

where \(\pi_i\) is the projection on the first or second component of a pair, and \(\mu(f: \tau \rightarrow \tau) \cdot e\) is an abstraction with a gradual type annotation and which uses \(f\) as a recursive binder. Function calls reduce according to the following rule

\[
\text{[App]} \; (\mu(f: \tau \rightarrow \tau) \cdot x \cdot e) \; e' \rightarrow e \; [f := \mu(f : \tau \rightarrow \tau) \cdot x \cdot e] \left[ x := e' \right]
\]

Although we do not provide an algorithmic way to compile the source language to the cast language, it is a task that can be derived from [6], using the following set of rules, where \(\Gamma \vdash e : \tau \rightsquigarrow E\) is the proposition which says that \(e\) of type \(\tau\) can be compiled into \(E\) of type \(\tau\) in the type environment \(\Gamma\).

\[
\begin{align*}
\text{[Const]} & \quad \frac{}{\Gamma \vdash c : b_c} \\
\text{[Var]} & \quad \frac{\Gamma \vdash x : \tau \rightsquigarrow E \quad \Gamma \vdash e_1 : \tau_1 \rightsquigarrow E_1 \quad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow E_2} {\Gamma \vdash e : \tau \rightsquigarrow E[\sigma_1 : \tau_1 \rightarrow \tau_2]} \\
\text{[FVar]} & \quad \frac{\Gamma \vdash e : \tau \rightsquigarrow E \quad \tau' \leq \tau} {\Gamma \vdash e : \tau \rightsquigarrow E \quad \tau' \leq \tau}
\end{align*}
\]

Fig. 8: Compilation to the Cast Language

The \([\text{Materialize}]\) rule, as explained in the footnote 2, creates type-cast expression. It uses two types, one for the main expression, and the other, at the index, for an eventual argument if the main expression is a function. This gives more precision to the type-checker.

**Definition 6.** The function \(\text{dom} : \text{Types} \rightarrow \text{Types}^+\) is defined as follows:

\[
\begin{align*}
\text{dom}(?) & = 1 \\
\text{dom}(\tau) & = \bigwedge_{\text{it} \in \text{je} \in \text{ji}} \wedge \sigma_j \quad \text{if } \tau = \bigvee_{\text{it} \in \text{je} \in \text{ji}} \left( \wedge \sigma_j \rightarrow \tau_j \right) \wedge \left( \wedge_{n \in \text{ni}} \neg(\sigma_n \rightarrow \tau_n) \right) \\
\text{dom}(\tau) & = \bot \quad \text{otherwise}
\end{align*}
\]

**Definition 7.** The result operator \(\circ\) is defined for types \(\tau\) and \(\sigma\) as:

\[
\tau \circ \sigma = \bigvee_{\text{it} \in \text{it}} \left\{ \bigvee_{Q \subseteq P_i \text{ s.t. } \sigma \notin \bigcap Q} \left( \bigwedge_{\text{je} \in \text{ji}} \wedge \rho \right) \right\}
\]

This definition achieves the goal of having \(\tau \circ \sigma\) reach the minimum of the set: \(\{\tau' \mid \tau \leq \sigma \rightarrow \tau'\}\)

**Definition 8.** The \(i\)-th projection is defined as:

\[
\pi_i \tau = \bigvee_{p \in \text{N} \subseteq \text{N}_p} \left( \bigvee_{\text{je} \in \text{jp}} \left( \bigwedge_{\text{aj} \in \text{aj}} \neg \tau_{n}^j \wedge \right) \right)
\]

With this definition, it can be proven that \(\pi_{i} \tau\) is the minimum of the set \(\{\tau' \mid \tau \leq \tau' \times 1\} \) (for \(i = 1\)) and \(\{\tau' \mid \tau \leq 1 \times \tau'\} \) (for \(i = 2\)).

### FORMAL SEMANTICS

#### B.1 Small Step

The small-step semantics of the cast language is defined in terms of values (ranger over by \( V \)), and contexts (ranged over by \( E \)), with the corresponding \([\text{CONTEXT}]\) and \([\text{CTXFAIL}]\) reduction rules:

\[
\begin{align*}
U & ::= \circ \mid \mu^x \cdot E \\
V & ::= k \mid U \mid U \langle \tau \rangle \mid (V, V) \mid (V, V) \langle \tau \rangle \\
E[] & ::= \square \mid E \mid V \mid (E, E) \mid (V, V) \mid \pi_1 E \mid E\langle \tau \rangle \\
\end{align*}
\]

\[
\begin{align*}
\text{Fig. 9: Small Step Reduction Rules}
\end{align*}
\]

The apply operator allows us to apply built-in functions (or functional constants) to values in our system. In order to respect type preservation, it should respect the condition:

\[
\forall V \forall V \quad (\text{type } \circ = \tau_1 \to \tau_2 \text{ and } \circ \vdash V : \tau_1) \implies \circ \vdash \text{apply}(\circ, V) : \tau_2
\]

The reduction rules are defined in Figure 9.

#### B.2 Big Step

The full set of rules for the big step semantics is given Figure 10.

### TYPE SYSTEM

The rules of Figure 11 describe a full declarative type system for the cast language.

\[
\begin{align*}
\text{Fig. 11: Typing Rules for the Cast Language}
\end{align*}
\]
A space-efficient call-by-value virtual machine for gradual set-theoretic types

Fig. 10: Big Step Semantics for the Cast Language

**D  TYPE PRESERVATION**

The cast mechanism for constants requires to prove the following lemma:

**Lemma 4.** $\forall b \in B \quad \forall \tau \quad \begin{cases} b \leq \tau^\emptyset & \Rightarrow b \land \neg \neg \leq \tau \\ b \leq (-\tau)^\emptyset & \Rightarrow b \land \neg \neg \leq -\tau \end{cases}$

**Proof.** Since $b \leq \tau^\emptyset$, we know that $\tau$ is not an arrow type or a pair type. Therefore, a simple proof by induction on the syntax of $\tau$ yields the result:

1. If $\tau = b'$ for $b' \in B$, since there are no unknown types in base types then $\tau^\emptyset = \tau$, hence $b \land \neg \neg \leq b \leq \tau^\emptyset = \tau$

The same result holds true for $-\tau$:

2. $b \land \neg \neg \leq b \leq -\tau^\emptyset = -\tau$

3. If $b \leq \tau_1 \land \tau_2^\emptyset = \tau_1^\emptyset \land \tau_2^\emptyset$, then $b \leq \tau_1^\emptyset$ and $b \leq \tau_2^\emptyset$, hence by induction $b \land \neg \neg \leq \tau_1$ and $b \land \neg \neg \leq \tau_2$ and therefore $b \land \neg \neg \leq \tau_1 \land \tau_2$.

4. If $b \leq \tau_1 \lor \tau_2^\emptyset$, then either $b \leq \tau_1^\emptyset$ or $b \leq \tau_2^\emptyset$ because $b$ is a basic type. Hence, by induction, either $b \land \neg \neg \leq \tau_1$ or $b \land \neg \neg \leq \tau_2$ and therefore $b \land \neg \neg \leq \tau_1 \lor \tau_2$.

5. If $b \leq -\tau$, then either $\tau$ is a basic type, a case which was already treated, or $\tau = \tau_1 \land \tau_2$, in which case

\[ (-\tau)^\emptyset = (-\tau_1 \land \neg \neg \tau_2)^\emptyset = (-\tau_1) \land (-\tau_2)^\emptyset \]

and this case is treated similarly to the case (3), using the induction hypothesis for the negation type. The last case, $\tau = \tau_1 \lor \tau_2$, yields:

\[ (-\tau)^\emptyset = (-\tau_1 \lor \neg \neg \tau_2)^\emptyset = (-\tau_1) \land (-\tau_2)^\emptyset \]

and it is treated similarly as the case (2).

**Lemma 5.** The following typing rule is derived by induction on $E$:

\[
\frac{}{\Gamma, x : \tau_2, E_1 : \tau_1 \quad E \vdash x \in \Gamma} {\text{Subst}} \quad \Gamma \vdash E_1[x := E_2] : \tau_1 \quad \Gamma \vdash E_2 : \tau_2}
\]

\[ \Gamma \vdash E : \tau \quad \text{and} \quad E \Rightarrow E' \quad \text{then} \quad \Gamma \vdash E' : \tau. \]

Proof. By induction on $E$, and by case on the reduction rule applied to obtain $E'$. Let $\Delta$ be a derivation tree for $\Gamma \vdash E : \tau$. We first prove type preservation in the case where $\Delta$ does not end with the rule [SUBSUME], after which we will prove that this assumption is safe.

1) Suppose that $\Delta$ does not end with the rule [SUBSUME].

- [BETA] $(\mu E : f \in \cdot X \cdot E) V \Rightarrow E[f := \mu_{E \in \cdot X \cdot E} \cdot X]$ by hypothesis and inversion of rule [APP], we have $\Gamma \vdash E : \tau$ and $\Gamma \vdash f : \tau \Rightarrow \cdot X \cdot E$. Double application of the rule [SUBST] in Lemma 5, since $\Gamma \vdash E : \tau$ and $\Gamma \vdash \mu_{E \in \cdot X \cdot E} \cdot X$, we obtain $\Gamma \vdash E[f := \mu_{E \in \cdot X \cdot E} \cdot X] : \tau$.

- [BETA] $(U \in \cdot \tau \cdot V \leftarrow (U \tau)_{\tau \in \cdot \tau \cdot \tau}) \langle \tau \rangle_{\tau \in \cdot \tau \cdot \eta}$ with type $U \in \cdot \tau \cdot \tau$, $\tau_{\in \cdot \tau \cdot \tau} = \tau_{\in \cdot \tau \cdot \tau}$, and $\eta_{\in \cdot \tau \cdot \tau} = (\tau_{\in \cdot \tau \cdot \tau} \leftarrow \tau_{\in \cdot \tau \cdot \eta}) \circ V$. By inversion of the typing rule [APP], we have $\Gamma \vdash U(\cdot \tau \cdot \tau) : \tau \Rightarrow \cdot \tau \cdot \tau$ and $\Gamma \vdash V : \tau$, implying that $\tau_{\in \cdot \tau \cdot \tau}$ and type $V \in \cdot \tau \cdot \omega$. By inversion on the typing rules for $\Gamma \vdash U(\cdot \tau \cdot \tau) : \tau \Rightarrow \cdot \tau \cdot \tau$, we have $\Gamma \vdash U : \tau_{\in \cdot \tau \cdot \tau}$ and $\Gamma \vdash \tau : \tau$. Then:

- By applying [CAST] with $\Gamma \vdash U : \tau_{\in \cdot \tau \cdot \tau}$, we have $\Gamma \vdash V(\cdot \tau \cdot \tau)_{\tau \in \cdot \tau \cdot \tau} : \tau_{\in \cdot \tau \cdot \tau}$, and by [SUBSUME], since $\tau_{\in \cdot \tau \cdot \tau} = \tau_{\in \cdot \tau \cdot \tau}$, we have $\Gamma \vdash V(\cdot \tau \cdot \tau)_{\tau \in \cdot \tau \cdot \tau} : \tau_{\in \cdot \tau \cdot \tau}$.

- By applying [APP], since $\Gamma \vdash U : \tau \Rightarrow \cdot \tau \cdot \tau$, we have $\Gamma \vdash U(\cdot \tau \cdot \tau)_{\tau \in \cdot \tau \cdot \tau} : \tau_{\in \cdot \tau \cdot \tau}$.

Finally, by rule [CAST], we have $\Gamma \vdash (U(\cdot \tau \cdot \tau)_{\tau \in \cdot \tau \cdot \tau})_{\tau \in \cdot \tau \cdot \tau} \circ V$. By property of the typing rules, we have $\Gamma \vdash U : \tau_{\in \cdot \tau \cdot \tau}$ and $\Gamma \vdash \tau : \tau$. Hence $\Gamma \vdash U : \tau_{\in \cdot \tau \cdot \tau}$ and $\Gamma \vdash \tau : \tau$. Therefore, by rule [CAST], we have $\Gamma \vdash U(\cdot \tau \cdot \tau)_{\tau \in \cdot \tau \cdot \tau} : \tau_{\in \cdot \tau \cdot \tau}$, which concludes by subsumption.

2) If the last case of $\Delta$ is [SUBSUME], it means that there is $\tau_{\in \cdot \tau \cdot \tau}$ such that $\tau_{\in \cdot \tau \cdot \tau} \Rightarrow \cdot \tau$ and that $\Delta'$ defined as $\Delta$ minus its last subsumption typing rules proves $\Gamma \vdash E : \tau_{\in \cdot \tau \cdot \tau}$ without using [SUBSUME] as its last rule. We use $\Delta'$ to prove that $\Gamma \vdash E' : \tau_{\in \cdot \tau \cdot \tau}$, from which we apply subsumption, since $\tau \in \cdot \tau \cdot \tau \Rightarrow \cdot \tau$ to obtain $\Gamma \vdash \tau : \tau$.

$\Box$

Lemma 7. (Progress) For every term $E$ such that $\emptyset \vdash E : \tau$, one of the following cases holds:

- there exists a value $V$ such that $E = V$
- there exists a term $E'$ such that $E \equiv E'$
- or $E \rightarrow\rightarrow \emptyset$

Proof. By complete induction over the expression $E$.

Case x. Impossible by hypothesis since a single variable cannot be well-typed in the empty environment.

Case c. Immediate since $\emptyset$ is a value.
In this section, we consider a version of our virtual machine that uses general variable identifiers instead of De Bruijn indices. It is essentially $\forall E \text{ SOUNDNESS}$.

Case $E_1, E_2$. By inversion of the typing rules, we deduce that $\forall \triangleright E_1 : \tau_1 \rightarrow \tau_2$ and $\forall \triangleright E_2 : \tau_1$. We can thus apply the induction hypothesis on both $E_1$ and $E_2$, which yields the following subcases:

- $\exists E'$ such that $E_1 \Leftrightarrow E_1'$. By rule [Context] and since $\Box E_2$ is a valid reduction context, $E_1 \Leftrightarrow E_1' \Leftrightarrow E_2$.

- $E_1 \Leftarrow \text{Fail1}$. By rule [CtxFail] and since $\Box E_2$ is a valid reduction context, $E_1 \Leftarrow \text{Fail1}$.

- $E_1$ is a value and $\exists E'$ such that $E_1 \Leftrightarrow E_1'$. Since $E_1$ is a value, $E_1 \Box$ is a valid reduction context, thus $E_1 \Leftrightarrow E_1' \Leftrightarrow E_1 E_2$ by rule [Context].

- $E_1$ is a value and $E_2 \Rightarrow \text{Fail1}$. Since $E_1$ is a value, $E_1 \Box$ is a valid reduction context, thus $E_1 E_2 \Rightarrow \text{Fail1}$ by rule [CtxFail].

- Both $E_1$ and $E_2$ are values. Reasoning by case analysis on $E_1$ and not considering ill-typed cases:

  - $E_1 = \mu x. f x. E$. Immediate since $\mu x. f x. E$ is a value.

  - By cases on $E$ yielding the following subcases:

    - $E_1 = U(\tau_i) \tau_i$ where $\tau_i \leq \tau \rightarrow \tau_2$. Then, by rule [Beta], $E_1 E_2$ reduces to $U(V(\tau_a) \downarrow \tau_a)(\tau_r) \downarrow \tau_r$ where $\tau = \tau_3 \rightarrow \tau_4$, $\tau_a = \tau_i \downarrow \tau_3$ and $\tau_r = (\tau_i \downarrow (\tau_3 \rightarrow \tau_4)) \circ \text{type } V$.

    - $E_1 = \Box$, then $E_1 E_2$ reduces to $\text{apply}(\Box, E_2)$.

Case $E_1, E_2$. By inversion of the typing rule [Pair], we deduce that $\forall \triangleright E_i : \tau_i$, for $i \in \{1, 2\}$. Thus, we can apply the induction hypothesis to $E$, yielding the following subcases:

- $E \Leftrightarrow E'$. Since $\Box E_1$ is a valid reduction context, $E_1 E_2 \Leftrightarrow (E_1, E_2)$ by rule [Context].

- $E \Leftarrow \text{Fail1}$. Since $\Box E_1$ is a valid reduction context, $E_1 E_2 \Leftarrow \text{Fail1}$ by rule [CtxFail].

- $E_2$ is a value and $E_1 \Leftrightarrow E_1'$. Since $E_2$ is a value, $(\Box E_2)$ is a valid reduction context, thus $E_1 E_2 \Leftrightarrow (E_1', E_2)$ by rule [Context].

- $E_2$ is a value and $E_1 \Leftarrow \text{Fail1}$. Since $E_2$ is a value, $(\Box E_2)$ is a valid reduction context, thus $E_1 E_2 \Leftarrow \text{Fail1}$ by rule [CtxFail].

- Both $E_1$ and $E_2$ are values. In this case, $(E_1, E_2)$ is itself a value, concluding this case.

Case $\pi \downarrow E$. By inversion of the typing rule [Proj], we deduce that $\forall \triangleright E : \tau_1 \times \tau_2$. Thus, we can apply the induction hypothesis to $E$, yielding the following subcases:

- $E \Leftrightarrow E'$. Since $\Box E$ is a valid reduction context, $\pi E \Leftrightarrow \pi E'$ by rule [Context].

- $E \Leftarrow \text{Fail1}$. Since $\Box E$ is a valid reduction context, $\pi E$ reduces to $\text{Fail1}$ by rule [CtxFail].

- $E$ is a value. By cases on $E$, not considering the ill-typed cases:

  - $E = (V_1, V_2)$. In this case, $\pi_1 E$ reduces to $V_1$ by rule [Proj].

  - $E = V(\tau_1) \tau_1$ where type $V \leq \downarrow \times \downarrow$. Then $\pi_1 E$ reduces to $(\pi_1 V) (\pi_1 \tau_1) \text{dom}(\pi_1 \tau_1)$ by rule [CastProj].

  - $E = V(\tau_1) \tau_1$ where type $V \leq \downarrow \times \downarrow$. Then $\pi_1 E$ reduces to $\text{Fail1}$ by rule [ProjFail].

Case $E(\tau_1) \tau_1$. By inversion of the typing rule [Cast], we deduce that $\forall \triangleright E : \tau$. Therefore, we can apply the induction hypothesis to $E$, yielding the following subcases:

- $E \Leftrightarrow E'$. Since $\Box (\tau_1) \tau_1$ is a valid reduction context, $E(\tau_1) \tau_1 \Leftrightarrow E'(\tau_1) \tau_1$ by rule [Context].

- $E \Leftarrow \text{Fail1}$. Since $\Box (\tau_1) \tau_1$ is a valid reduction context, $E(\tau_1) \tau_1 \Leftarrow \text{Fail1}$ by rule [CtxFail].

- $E$ is a value. This yields the following subcases:

  - $E = k$. If $\exists (k) \leq \tau_1 \downarrow$, then $E$ reduces to $\text{Fail1}$ by rule [Cast2]. Else, $E$ reduces to $\text{Fail1}$ by rule [CastFail].

  - $E = U$. Then $E(\tau_1) \tau_1$ is a value.

  - $E = U(\tau_1) \tau_1$ and $\tau_2 = \downarrow$. Then $E(\tau_1) \tau_1$ reduces to $\text{Fail1}$ by rule [CastFailU].

  - $E = U(\tau_1) \tau_1$ and $\tau_2 \neq \downarrow$. Then $E(\tau_1) \tau_1$ reduces to $E(\tau_1 \land \tau_3) \tau_1 \land \tau_4$ by rule [CastU].

\[ \square \]

**Theorem 9. (Soundness)** For every term of the cast language $E$ such that $\forall \triangleright E : \tau$, either there exists a value $V$ such that $E \Leftarrow V$, or $E \Rightarrow \text{Fail1}$, or $E$ diverges.

**Proof.** By Corollary of Lemma 7 and Lemma 6.\[ \square \]

### E SOUNDNESS

In this section, we consider a version of our virtual machine that uses general variable identifiers instead of De Bruijn indices. It is essentially the same, except that the result is easier to state, as well as the interpretation between big step values and machine values.

**Definition 2. (Interpretation)** We define $\llbracket E \rrbracket$ an interpretation from the cast language to the machine elements. It defines a bijection of type $\text{Val} \Rightarrow \text{MVal}$ from cast values to machine values:

\[
\begin{align*}
\llbracket c \rrbracket & = c \\
\llbracket \mu x. f x. E, \langle \tau_1 \rangle \tau_2 \rangle \rrbracket & = [x, f, T [E], [E], \langle \tau_1 \rightarrow \tau_2 \rangle \tau_1] \\
\end{align*}
\]

It also extends to environments: $\forall \exists E \in \text{dom}(E), [E](x) = [E](x), \text{and } [E]$ is undefined for other values.

**Theorem 10.** For all term of the cast language $E$,

\[
\forall E (E \Rightarrow v \quad \Rightarrow \forall (c, s, d) (C [E] : c, [E], s, d) \Rightarrow (c, [E], [v], s, d))
\]
LEMMA 8. For all term of the cast language $E$, 

$$ \forall E \left( \vdash E \Rightarrow v \quad \Rightarrow \quad \forall (c, s, d) \quad \left( C[E] ; c, [E], s, d \Rightarrow^* (c, [E], [v], s, d) \right) \right) $$

and with d framed

$$ (T[E], [E], s, d) \Rightarrow^* (\text{ret}, [E], [v], s, d) $$

Proof. Proving Lemma 2 by induction on $E$. First considering $C[E]$:

- if $E = c$. For all $E$, if $\vdash c \Rightarrow v$, then we have $v = c$, and for all $(c, s, d)$, since $C[c] = push\ c,$

$$ \left( C[c] ; c, [E], s, d \right) = \left( push\ c; c, [E], s, d \right) \quad \rightarrow \quad \left( c, [E], [c], s, d \right) $$

- if $E = x$. For all $E$, if $\vdash x \Rightarrow v$ then $x \in \text{dom}(E)$ and $v = E(x)$, which means that for all $(c, s, d)$,

$$ \left( push\ x; c, [E], s, d \right) \quad \rightarrow \quad \left( c, [E], [E(x)], s, d \right) $$

- if $E = \mu^{\tau_1 \rightarrow \tau_2} f x. E'$. For all $E$, if $\vdash E \Rightarrow [\mu^{\tau_1 \rightarrow \tau_2} f x. E', \langle \tau_1 \rightarrow \tau_2 \rangle^*]$, then for all $(c, s, d)$,

$$ \left( push\ (x, f, T[E']), \langle \tau_1 \rightarrow \tau_2 \rangle^* ; c, [E], s, d \right) \quad \rightarrow \quad \left( c, [E], [x, f, T[E'], \langle \tau_1 \rightarrow \tau_2 \rangle^*], s, d \right) $$

- if $E = E' \langle \tau_1 \rangle^*$. For all $E$, if $\vdash E' \Rightarrow [\mu^{\tau_1 \rightarrow \tau_2} f x. E', \langle \tau_1 \rangle^*]$, then for all $(c, s, d)$,

$$ \left( push\ (\langle \tau_1 \rangle^* ; C[E'], \text{cast} ; c, [E], s, d \right) \quad \rightarrow \quad \left( C[E'] ; \text{cast} ; c, [E], \langle \tau_1 \rangle^*, s, d \right) $$

$$ \rightarrow^* \left( \text{cast} ; c, [E], [v'], \langle \tau_1 \rangle^*, s, d \right) \quad \text{by induction hypothesis on } E' \text{ with } (c', s', d') = (\text{cast} ; c, \langle \tau_1 \rangle^*, s, d) $$

$$ \rightarrow \left( c, [E], [v'], s, d \right) \quad \text{since } \text{cast}(v', \langle \tau_1 \rangle^*) = v $$

- if $E = E_1 E_2$. Let $E$ such that $\vdash E \Rightarrow v'$. This expression evaluates using one of the two big step semantics rules for function application:

- Using $\beta^*$. By inversion, we have $\vdash E_1 \Rightarrow v_1 = [\mu^{\tau_1 \rightarrow \tau_2} f x. E, E', \langle \tau_3 \rangle^*]$ and $E + E_2 \Rightarrow v_2$. Therefore,

$$ \left( C[E_1] ; C[E_2] \text{ app} ; c, [E], s, d \right) $$

$$ \rightarrow^* \left( C[E_2] ; \text{app} ; c, [E], [v_1], s, d \right) \quad \text{by induction hypothesis on } E_1 $$

$$ \rightarrow^* \left( \text{app} ; c, [E], [v_2], [v_1], s, d \right) \quad \text{by induction hypothesis on } E_2 $$

$$ \rightarrow \left( T[E'], [E'(x := v_2 f := v_1)], s, (c, [E]).d \right) \quad \text{since } [v_1] = [x, f, T[E'], [E'], \langle \tau_3 \rangle^*] $$

$$ \rightarrow \left( \text{ret}, [E'(x := v_2 f := v_1)], [v].s, (c, [E]).d \right) \quad \text{by induction hypothesis on } E'(x := v_2 f := v_1) \Rightarrow E' \Rightarrow v' $$

$$ \quad \text{and since } (c, [E]).d \text{ is a framed dump} $$

- Using $\beta_2$. By inversion, we have:

$$ E \vdash E_1 \Rightarrow v_1 = [\mu^{\tau_1 \rightarrow \tau_2} f x. E, E', \langle \tau_3 \rangle^*] $$

$$ E \vdash (v_4 v_3)\langle \tau_1 \circ \text{type}(v_2) \rangle^{\text{dom } \tau_1 \circ \text{type}(v_2)} \Rightarrow v $$

$$ \text{with } v_4 = [\mu^{\tau_1 \rightarrow \tau_2} f x. E', E', \langle \tau_3 \rangle^*] $$

Hence
We now consider a space-efficient call-by-value virtual machine for gradual set-theoretic types.

\[
\begin{align*}
(C[E_1]; C[E_2]; \text{app}; c, [E], s, d) \\
\rightarrow^* (C[E_2]; \text{app}; c, [E], [v_1], s, d) & \quad \text{by induction hypothesis on } E_1 \\
\rightarrow^* (\text{app}; c, [E], [v_2], [v_1], s, d) & \quad \text{by induction hypothesis on } E_2 \\
\rightarrow (\text{cast}; \text{app}; \text{cast}; c, [E], [v_2], \tau_2 \text{dom } \tau_1, [v_4], \tau_2 \text{ dom } \tau, s, d) & \quad \text{with } \tau = \tau_1 \circ \text{type } v \\
\rightarrow (\text{app}; \text{cast}; c, [E], [v_3], [v_4], \tau_2 \text{ dom } \tau, s, d) \\
\rightarrow (T[E'], [E'(x := v_3 f := v_4)], \tau_2 \text{ dom } \tau, s, \text{cast}; c, [E]) \cdot d) & \quad \text{by induction on } E' (x := v_3 f := v_4) \Rightarrow v \\
\rightarrow (\text{ret}, [E'(x := v_3 f := v_4)], [v], \tau_2 \text{ dom } \tau, s, \text{cast}; c, [E]) \cdot d) & \quad \text{and since } \text{cast}; c, [E] \cdot d \text{ is a framed dump} \\
\rightarrow (\text{cast}; c, [E], [v], \tau_2 \text{ dom } \tau, s, d) \\
\rightarrow (c, [E], [v], s, d)
\end{align*}
\]

We now consider \( T[E] \):

- If \( E = E_1 E_2 \). For all \( E \), if \( E \vdash E_1 \Rightarrow v \) then we have \( E \vdash E_1 \Rightarrow v_1, E \vdash E_2 \Rightarrow v_2, \) and \( C[E] = C[E_1]; C[E_2] \cdot \text{tap} \). Let \((s, d)\) with \( d \) having a stack frame on top of it.

\[
(C[E_1]; C[E_2]; \text{tap}; c, [E], s, d) \\
\rightarrow^* (C[E_2]; \text{tap}; c, [E], [v_1], s, d) & \quad \text{by induction on } E \vdash E_1 \Rightarrow v_1 \\
\rightarrow^* (\text{tap}; c, [E], [v_2], [v_1], s, d) & \quad \text{by induction on } E \vdash E_2 \Rightarrow v_2
\]

- If \([v_1] = [x, f, T[E'], [E'], (\tau_1)\tau_2]^*\), then the tail call “tap” is applied using the untyped machine transition \( t\beta_4 \):

\[
(T[E'], [E'(x := v_2 f := v_1)], s, d) \\
\rightarrow (\text{ret}, [E'(x := v_2 f := v_1)], [v], s, d) & \quad \text{by induction on } E'(x := v_2 f := v_1) \Rightarrow v \\
& \quad \text{and since } d \text{ is framed}
\]

- Else if \([v_1] = [T[E'], [E'], (\tau_1)\tau_2]^\rho\), then the typed machine transition \( t\beta_2 \) is applied, and by inversion, since the corresponding big step rule is \( [\beta_3] \), we have:

\[
E \vdash v_2 (\tau_2 \text{ dom } \tau_1) \Rightarrow v_3 \quad E \vdash (v_4 v_3) (\tau_1 \circ \text{type}(v_2)) \text{dom } \tau_1 \circ \text{type}(v_2) \Rightarrow v \quad \text{with } v_4 = [\mu f \xrightarrow{\tau_2} f x . E', E', (\tau_3)\tau_1]^* 
\]

Hence the successive configurations:
\[
\left( \text{cast} ; \text{tca} \left( \langle \tau \rangle^\text{dom} r ; c , [E] , [v_2], \tau_2^\text{dom} n , [u_4], s , d \right) \right) \quad \text{with } \tau = \tau_1 \circ \text{type} (v_2)
\]

\[
\rightarrow \left( \text{tca} \left( \langle \tau \rangle^\text{dom} r ; c , [E] , [v_2], [v_4], s , d \right) \right) \quad \text{by induction on } E \vdash v_2 (\tau_2)^\text{dom} r_2 \Rightarrow v_3
\]

\[
\rightarrow \left( \text{tap} , [E] , [v_3],[v_4], s , \text{dump}(\tau^\text{dom} r , d) \right) \quad \text{by induction on } E \vdash v_4 v_3 \Rightarrow v'
\]

\[
\rightarrow \left( \text{t} \beta^\ast \left( T' [E'] , [E'] (x := v_3 f := v_4) , s , \tau^\text{dom} r , d \right) \right) \quad \text{where dump}(\tau^\text{dom} r , d) = \tau^\text{dom} r . d \text{ since } d \text{ is framed}
\]

\[
\rightarrow \left( \text{ret} , [E'] (x := v_3 f := v_4) , [v'], s , \tau^\text{dom} r , d \right) \quad \text{by induction on } E \vdash v_4 v_3 \Rightarrow v'
\]

\[
\rightarrow R^\ast \left( \text{ret} , [E'] (x := v_3 f := v_4) , [v'], s , d \right) \quad \text{since } E \vdash v' \langle \tau \rangle^\text{dom} r \Rightarrow \nu
\]

- if \( E = (E_1 E_2) \langle \tau_1 \rangle^{\circ} \). Let \( E \) such that \( E \vdash (E_1 E_2) \langle \tau_1 \rangle^{\circ} \Rightarrow \nu \). By inversion, we have \( E \vdash E_1 E_2 \Rightarrow \nu' \) and \( E \vdash \nu' \langle \tau_1 \rangle^{\circ} \Rightarrow \nu \). Let \( (s , d) \) with \( d \) framed.

\[
\left( C[E_1] ; C[E_2] ; \text{tca} \langle \tau_1 \rangle^{\circ} ; c , [E] , s , d \right)
\]

\[
\rightarrow \ast \left( \text{tca} \langle \tau_1 \rangle^{\circ} ; c , [E] , [v_2],[v_1], s , d \right) \quad \text{by induction on } E_1 \text{ and } E_2
\]

- If \( [v_1] = [x , f , T [[E']] , [E'] , (\tau_1)^{\circ}] \ast \), then the function call \( \text{tca} \langle \tau_1 \rangle^{\circ} \) triggers a transition to the state:

\[
\left( T [E'] , [E'] (x := v_2 f := v_1) , s , \text{dump}(\tau_1)^{\circ} , d \right)
\]

\[
\rightarrow \ast \left( \text{ret} , [E'] (x := v_2 f := v_1) , [v'], s , \text{dump}(\tau_1)^{\circ} , d \right) \quad \text{by induction on } E \vdash E' \Rightarrow \nu'
\]

\[
\rightarrow R^\ast \left( \text{ret} , [E'] (x := v_2 f := v_1) , [v'], s , d \right) \quad \text{after cast application}
\]

- Else, if \( [v_1] = [x , f , T [[E']] , [E'] , (\tau_1)^{\circ}] \ast \), then the same reduction happens with supplementary casts upfront and when merging the argument of \( \text{tca} \langle \tau_1 \rangle^{\circ} \).

- The case of conditionals \( E = \text{if } E \text{E}_1 \text{E}_2 \) is trivial by induction, and other tail expressions can easily be obtained from the previous cases since their compiled code takes the form of \( T [[E]] = C[[E]] ; \text{ret} \).

\section{SPACE EFFICIENCY}

\begin{theorem}
Let \( E \) a cast expression. There exists a finite set of atoms \( \mathcal{A} \) such that for all \( E' \) such that, if \( E \longrightarrow^* E' \), then any type-cast in \( E' \) is represented by a BDD using exclusively atoms from \( \mathcal{A} \).
\end{theorem}

\begin{proof}
Every function interface (the type of \( \mu^\text{t \rightarrow r} f x . E \) in \( E \), and every type-cast expression is built using a finite distinct number of atoms. We denote by \( \mathcal{A} \) this set of atoms, and prove by induction that for \( n \in \mathbb{N} \), if \( E \) reduces to \( E' \) in \( n \) steps, written \( E \rightarrow_n E' \), then all the interface and type-casts that appear in \( E' \) are built using the atoms of \( \mathcal{A} \).

- \( n = 0 \) is true.
- \( n \in \mathbb{N}^* \). The only small step reduction rules that modify types are [BETAC] (typed beta-reduction), [CASTU] (type compression) and [CASTP] (apply cast on pairs). They use the operations \( \text{dom}, \circ, \land \), therefore they conserve the atoms used in the previous expression.
\end{proof}