Gradual Typing for Mutable Objects

Jeremy G. Siek\textsuperscript{1}, Michael M. Vitousek\textsuperscript{2}, and Shashank Bharadwaj\textsuperscript{1}

\textsuperscript{1} Department of Electrical, Computer, and Energy Engineering
\textsuperscript{2} Department of Computer Science
University of Colorado Boulder
Boulder, CO 80309, USA
\{jeremy.siek,michael.vitousek,shashank.bharadwaj\}@colorado.edu

Abstract. Gradual typing enables a mixture of static and dynamic typing within the same program. There are four properties that are desirable in a gradually-typed language: 1) the type system allows implicit conversions to and from the dynamic type so that values can easily pass between statically and dynamically-typed regions of code, 2) the overhead associated with implicit conversions is low, 3) if one of these conversions fails, the guilty source location is blamed, and 4) statically-typed regions of code execute just as efficiently as code written in a statically-typed language. This paper presents the first design, monotonic objects, that has all four properties in the presence of mutable state. On the way to this design we develop guarded objects, a natural extension of earlier work but that imposes some overhead on statically-typed regions of code.

1 Introduction

Despite the polarizing affect that static and dynamic typing has on programmers, the last decade has seen a renewed interest by researchers to integrate the two [Anderson and Drossopoulou, 2003, Flanagan et al., 2006, Tobin-Hochstadt and Felleisen, 2006, Wadler and Findler, 2009, Bierman et al., 2010, Wrigstad et al., 2010, Bayne et al., 2011]. This paper concerns the gradual typing approach to this integration [Siek and Taha, 2006, Herman et al., 2007, Ina and Igarashi, 2011, Wolff et al., 2011, Takikawa et al., 2012, Rastogi et al., 2012]. Attesting to the practical utility of gradual typing, three of Microsoft’s languages, Visual Basic, C\#, and TypeScript, provide gradual typing to some degree.

The goal of gradual typing is to let programmers enjoy the benefits of either typing discipline within different regions of a program. The statically typed regions should benefit from efficient execution and from the early detection of type-related bugs. The dynamically typed regions should benefit from the freedom that comes from the absence of static type checking. But there is a tension between these design goals, particularly in languages with mutable state. In this paper we explore two new points in the design space for mutable objects, including the first design that simultaneously enables the seamless integration of statically and dynamically-typed regions of code while preserving the efficient execution of statically-typed regions.
At the heart of gradual typing is a consistency relation on types, written \( T_1 \sim T_2 \), that allows implicit conversions to and from the dynamic type, written \( \ast \), while disallowing conversions between inconsistent types such as int and string [Siek and Taha, 2006]. The consistency relation is orthogonal to subtyping, and to support both subtyping and consistency one can compose the two relations [Siek and Taha, 2007]. The dynamic semantics of gradual typing is defined by a translation to an intermediate language with explicit casts.

The original treatment of mutable state by Siek and Taha [2006] leaves something to be desired because it did not allow for the passing of ML-style references between static and dynamic code. The consistency rule, shown to the right, required the underlying type \( T \) to be invariant. Programs could not pass a reference of type \( \text{ref int} \) to a function expecting \( \text{ref} \ast \). The semantics of Siek and Taha [2006] does enable the fast execution of statically-typed code. The canonical forms lemma only admits one kind of value per type, and correspondingly, there is only one reduction rule for each elimination form. Thus, there is no need to dispatch on the kind of value. However, there is another efficiency concern: the cost of passing values between static and dynamic regions. Every application of a cast to a function creates a wrapper, and a function may be cast many times causing an unbounded space leak.

Herman et al. [2007] solve both the space efficiency issue and the problem with casting references. They represent casts using the Coercion Calculus of Henglein [1994] in which sequences of coercions normalize to coercions of at most length 3. Herman et al. [2007] enable the casting of references by using the more permissive consistency rule shown on the right and by creating a new form of coercion, \( \text{ref} \ c_1 \ c_2 \), to represent casts between reference types. The coercion \( c_1 \) is used when writing and \( c_2 \) is used when reading, perhaps inspired by Reynolds [1997]. Unfortunately, the coercion-based implementation of gradual typing introduces overhead in the execution of statically-typed code. For example, a value of type \( \text{ref int} \) may be simply an address or it may be an address wrapped in a coercion. Thus, every dereference must dispatch on the kind of value at run-time.

While efficiency is important, so is support for debugging. The casts used in gradual typing are closely related to contracts. The checking of higher-order contracts (involving functions or objects) happens in a delayed fashion, so an error may occur far away from the static location of the contract. Findler and Felleisen [2002] connect contract violations back to their static location using a technique now called blame tracking. Siek et al. [2009] adapt blame tracking to the Coercion Calculus, thereby improving the diagnostics for cast errors in space-efficient implementations of gradual typing. However, the dispatching overhead in statically-typed code remains.

Wrigstad et al. [2010] remove this overhead by introducing a distinction between concrete types, for which there can be only one kind of value, and like types, for which there can be several kinds of values. However, their approach
Table 1. Summary of the gradual typing literature with respect to four properties.

Table 2. The two designs in this paper.
2 Challenges of Gradual Typing with Mutable Objects

We begin with a review of gradual typing and discuss the challenges of adding gradual typing to languages with mutable objects. These challenges motivate our two designs, guarded and monotonic objects, which we introduce through examples written in a gradually-typed variant of Python.

2.1 Gradual Typing and Mutable State

To briefly review gradual typing, we begin with an example in which a dynamically-typed object is passed to the statically-typed `insert_to_db` function:

```python
class Person(object):
    def __init__(self, id, name):
        self.id = id
        self.name = name

def insert_to_db(p : {id:int, name:str}) -> void:
    ... p.name ...

person = Person(1, 'John Doe')
insert_to_db(person)
```

At the call to `insert_to_db`, the `person` object is downcast from the dynamic type `⋆` (implicit in the source code) to the object type `{id:int, name:str}`. In this case the cast succeeds, but in general, such casts may fail. In the below example, `person2` has a unicode string in its `name` field, whose type is not consistent with `str`, so the implicit cast signals an error.

```python
person2 = Person(2, unicode('Pi'))
insert_to_db(person2) # Error, blame the implicit cast on this line
```

Next we consider an example in which gradual typing interacts in a problematic way with mutable state. (The problem is analogous to the classic problem of covariant subtyping for arrays.) Suppose that we create an alias to the dynamically-typed `person` object with the `typed_person` variable. Then we mutate the `name` field of `person` to a unicode string and finally call `insert_to_db`.

```python
typed_person : {id:int, name:str} = person
person.name = unicode(person.name)
insert_to_db(typed_person)
```

The problem is that the type of the value in the `name` field has changed out from under the `typed_person` alias. In a naive implementation of gradual typing, when the statically-typed `insert_to_db` function accesses `p.name`, the result would be a unicode string instead of a string, which represents a hole in the type system. The following subsections present two different solutions to this problem: guarded objects and monotonic objects.
2.2 Guarded Objects

In the guarded objects approach, each cast creates a guarded reference (a kind of proxy) to the underlying object. When fields are accessed through a guard, casts are applied to bridge the gap between the underlying type of the object and the type of the guard. In the problematic example above, the implicit cast in the assignment to `typed_person` on line 13 creates a guard with type `{id:int, name:str}`. The subsequent assignment of a unicode string to the `name` field succeeds because the type of `person.name` is `⋆`. Finally, the guarded reference is passed to `insert_to_db` which tries to access `p.name`. The guard handles this access by casting the unicode string to `str`, triggering a run-time error and blaming the cast on line 13. Thus, guarded objects close the hole in the type system by performing checks at every read and write to a member, analogous to how Java maintains type-safety despite the covariant subtyping of arrays. The idea of guarded objects is not novel but making them space-efficient in the presence of higher-order casts is an open challenge that we address in Sections 4 and 5.

The main disadvantage of guarded objects is that the guarding overhead occurs at every member access, even in statically-typed code. The following example sorts a list of persons by their id.

```python
def sort_persons_by_id(lst: list({id:int, name:str})) → void:
    . . . lst[i].id < lst[j].id . . .
person_list = [Person(1, 'John'), Person(2, 'Jane'), ...]
sort_persons_by_id(person_list)
```

Consider the task of separately compiling the statically-typed `sort_persons_by_id` function. The compiler does not know whether the elements in `lst` will be guarded or unguarded references and thus generates code that can handle either case.

2.3 Monotonic Objects

The proxies used by guarded objects are needed because the values in an object’s fields may become inconsistent with the types of the guards. We can remove the overhead from fully-static field accesses (i.e., `⋆` does not appear in their static type) by ensuring that the runtime type of an object is no more dynamic than any reference to the object. Thus, if there is a fully-static reference, then the type of the field’s value must exactly match the type of the reference. To maintain this invariant, casts cause an object’s type to become more specific.

Turning back to the list sorting example, we see that the `people_list` is cast to `list({id:int, name:str})` at the call to `sort_persons_by_id`. Rather than wrapping the list in a proxy, as occurs with guarded objects, now the elements are are cast to `{id:int, name:str}`. While casting all the elements of a list is a linear-time operation, any subsequent casts to the same list would be no-ops. So the worst-case overhead for the entire program is only a constant factor more than a cast-free execution of the program. Now consider the accesses to the `id` field inside the `sort_persons_by_id` function. The type of the field at this access is `int`, which is fully static, so by
our established invariant, the underlying value in the field must be an integer. Thus, the field access can be compiled to overhead-free code. So with monotonic objects, in regions of code that are fully static, we are able to achieve performance comparable to that of a statically-typed language.

Next we discuss how to handle non-static (partially to fully dynamic) member reads and writes. We return to the first example in which a dynamically typed person is passed to the statically-typed \texttt{insert\_to\_db} function. However, this time we read from \texttt{id} and write to \texttt{name} after the call to \texttt{insert\_to\_db}.

```python
21 def insert\_to\_db(p: \{id:int, name:str\}) \rightarrow void:
22 ....
23 person = Person(1, 'John Doe')
24 insert\_to\_db(person)
25 print(person.id)
26 person.name = unicode(person.name) \# Error: blame cast on line 24
```

When the person is passed to \texttt{insert\_to\_db}, the underlying object is modified to have type \{id:int, name:str\} by downcasting its members. But now we have a problem with the access to \texttt{id} after the call. The value in the \texttt{id} field is an integer, but the type system expects it to have type \texttt{*}. We resolve this issue by casting from \texttt{int} to \texttt{*} (boxing the integer) during the read. We emphasize that such casts are only needed when the type of the field at the read site is partially dynamic; if the type of the field is fully-static, then no casts need to be performed.

Next we turn to the assignment of a unicode string to the \texttt{name} field of the \texttt{person} object. In this case we cast the new value (a unicode string) from the field’s static type (which is \texttt{*}) to the run-time type of the value in the field, which is \texttt{str}. But this cast fails because the unicode string type is not consistent with \texttt{str}, and the cast on line 24 is blamed. Thus, we see that while monotonic objects are more efficient, they are also more restrictive than guarded objects.

Summary. We have discussed two designs for gradually-typed, mutable objects. Both approaches satisfy the fundamental goal of maintaining type safety in the face of mutable state and something akin to covariant subtyping. However, the characteristics and ideal use cases of each approach vary. The guarded objects approach, by virtue of its ability to support strong updates (changes in the runtime types), is appropriate for situations where the bulk of the program is written in a highly dynamic style. The monotonic objects approach, on the other hand, provides efficient field access in statically-typed code, which makes it ideal in situations where high performance is needed and where the intention of the developer is to migrate towards a fully-static program. The monotonic approach also deals nicely with interoperability between languages such as Python and Java. Jython (Python on the JVM [Hugunin et al., 1997]) currently uses an approach similar to guarded objects but programmers complain of the overhead. Using monotonic objects, this overhead could be removed which would enable high performance Java libraries to be used from Python without undue performance degradation.

In the rest of the paper, we develop mathematical models of both approaches.
3 Gradual typing and the $\text{impl} \rightarrow \lambda^*$-calculus

Figure 1 defines the types of the $\text{impl} \rightarrow \lambda^*$-calculus, which include the dynamic type $\star$ to support gradual typing and also some basic types (int and bool), function types, and object types of the form $[\Delta_1; \Delta_2]$. The $\Delta_1$ is a maps a field’s label to the field’s type and $\Delta_2$ maps a method’s label to its type.

Figure 1 also defines three relations on types: consistency, subtyping, and naive subtyping. The purpose of the the first two relations is to define which implicit conversions are allowed by the type system. Subtyping defines the safe up-casts, that is, conversions that never fail, whereas consistency allows any cast that might succeed. The naive subtyping relation captures the notion of how “dynamic” a type is, with $\star$ being the most dynamic type. The type $\star \rightarrow \text{int}$ is more dynamic than $\text{int} \rightarrow \text{int}$, so $\text{int} \rightarrow \text{int} \prec_\circ \star \rightarrow \text{int}$ even though these types are not in the subtype relation: $\text{int} \rightarrow \text{int} \not\prec_\circ \star \rightarrow \text{int}$ because $\star \not\prec_\circ \text{int}$. The naive subtyping relation is used to express invariants of the dynamic semantics.

The subtyping relation on object types allows width subtyping but not depth subtyping for both fields and methods because both are mutable. The naive subtyping relation differs from subtyping because it allows depth subtyping on objects and because parameters of functions are covariant (instead of contravariant). The consistency relation is more permissive than subtyping, allowing widening and narrowing of an object, as well as variation in depth. The consistency relation just requires the common fields and methods to have consistent types. Unlike either subtyping relation, the consistency relation is symmetric, so it does
not make sense to talk about covariance versus contravariance. Additionally, the consistency relation is non-transitive.

Figure 2 defines the syntax and type system of the impλ*-calculus. This calculus includes some constants (Booleans and integers) and primitive operations, lambda expressions, let expressions, explicit casts, and objects with field and method projection and assignment. The calculus does not provide a special syntactic form for methods, but instead uses lambda expressions whose parameter is bound to the “self” argument during a method call. Method calls, which have the form \( e \rightarrow t() \), are syntactic sugar for method projection plus self application. Multi-parameter methods can be simulated via currying.

The type system in Figure 2 includes the standard subsumption rule, so implicit up-casts are allowed according to the subtyping relation. The gradual nature of the type system can be seen in the rules that mention the dynamic type \( \star \) and the consistency relation \( \sim \). If an expression has type \( \star \), then you can call it with an argument of any type. Also, you can project from and assign to any field or method of something of type \( \star \).

The consistency relation is used where one might expect to see type equality. In a function application where the function has type \( T_1 \rightarrow T_3 \), the parameter type \( T_2 \) does not have to be equal to \( T_1 \), we just require \( T_1 \sim T_2 \). Except for subsumption, this type system is syntax directed, so it is straightforward to implement a type checking algorithm. (Siek and Taha [2007] describe an algorithm that handles subsumption in a gradual type system.)

As alluded to in the introduction, the dynamic semantics of a gradually-typed language is given by a translation to an intermediate language with explicit casts. For example, Ahmed et al. [2011] use the notation \( e : T_1 \Rightarrow^{\ell} T_2 \) for casting expression \( e \) from its static type \( T_1 \) to \( T_2 \). The \( \ell \) is the blame label that uniquely identifies the static location of the cast. In this paper we will use a variant of the Coercion Calculus to express casts and achieve space efficiency, so our notation for explicit casts in the intermediate language is \( e : c \) where \( c \) is a coercion. We discuss our addition of object coercions in the next section.

We use slightly different intermediate languages for guarded objects and monotonic objects. Both include explicit casts, but the intermediate language for monotonic objects includes a syntactic distinction between fast, statically-typed member projections versus dynamically-typed member projections. Thus, the intermediate language for guarded objects is defined in Section 5 and the intermediate language for monotonic objects is in Section 6.

4 Coercions for Mutable Objects

We begin with a review the Coercion Calculus and then develop object coercions.

The Coercion Calculus defines a set of instructions (or combinators) that tell a machine how to convert a value of one type to another type. Figure 3 defines the syntax of the Coercion Calculus extended with object coercions. The simplest coercion is the identity coercion, written \( \iota \), which does not change the type of the coerced value. The injection coercion \( I! \) converts from type \( I \) to \( \star \). One can
think of this coercion as “boxing” its input. The category $I$ of injectable types includes all the types except $\star$. The evil twin of injection is projection, written $I^\tau$, which converts from $\star$ to the type $I$. This is a dangerous operation and might fail, in which case the blame goes to label $l$. In the original Coercion Calculus, the injectable types did not include arbitrary function types, but only $\star \to \star$. The approach we take here, of including arbitrary function (and object) types in the injectable types is a characteristic of the D blame tracking strategy of Siek et al. [2009]. The advantage of this approach is that safe casts are characterized by the subtyping relation that one would expect, that of Figure 1.

In some situations, to conserve space, we normalize coercions before it is time to apply them to a value. In such cases, we do not immediately signal an error, but instead just record it. This is the role of the Fail$^I$ coercion [Herman et al., 2007]. The coercion $c_1 \to c_2$ is a function coercion, which applies $c_1$ to the argument of a function and $c_2$ to the return value. The sequence coercion $c_1 : c_2$ applies $c_1$ and then $c_2$. (The standard notation for sequencing is $c_2 \circ c_1$, but we use the semicolon to carry on the improved left-to-right reading of casts.)
The type system for coercions is defined in Figure 3. We write $\vdash c : T_1 \Rightarrow T_2$ to say $c$ is a well-typed coercion from source type $T_1$ to target $T_2$. Figure 3 also defines a partial function $\langle T_1 \Rightarrow T_2 \rangle^\ell$ that creates a coercion from a source and target type and a blame label. This function is only defined when $T_1 \sim T_2$. When the source and target are subtypes, $T_1 <: T_2$, the resulting coercion contains no blame labels, so it is straightforward to see why such casts are safe.

### 4.1 Efficient Normalization of Coercions

The dynamic semantics of coercions is defined by a reduction semantics. However, there is a gap between such a semantics and an implementation. Following recent work of Siek and Garcia [2012], we define a composition operator, written $c_1 \# c_2$, that takes two coercions in normal form and produces the normal form for sequencing the two coercions (Figure 4). The composition operator is closely related to, but different from, the coercion sequence constructor $c_1; c_2$. The coercion $c_1; c_2$ normalizes to $c_3$ if and only if $c_1 \# c_2 = c_3$.

Let us consider some examples of how coercions work. Suppose the integer 42 is cast to dynamic. The resulting value has the form $42 : \text{int}$!. Next, suppose this value is cast back to $\text{int}$, that is, it is projected using the coercion $\text{int}^?\ell$. The semantics will compose the coercion on the value with the coercion of the cast. In this case, the result is the identity coercion: $\text{int}! \# \text{int}^?\ell_1 = \langle \text{int} \Rightarrow \text{int} \rangle^?\ell_1 = \iota$. Thus, the result of the cast back to int will simply be 42. On the other hand, if we cast to an inconsistent type, such as $\text{bool}$, the result is a failure coercion which would then trigger a cast error: $\text{int}! \# \text{bool}^?\ell_2 = \text{Fail}^?\ell_2$.

Our composition operator implements eager cast checking [Herman et al., 2007, Siek et al., 2009], which means that a cast failure is reported as soon as it is detected and applied to a value. With respect to the composition operator, this means that if the composition of any sub-coercion results in $\text{Fail}^?\ell$, then the overall result is $\text{Fail}^?\ell$. For example, the composition of the following two function coercions produces a failure because the composition of the codomain coercions produces a failure: $(\iota \rightarrow \text{int}!) \# (\iota \rightarrow \text{bool}^?\ell_1) = \text{Fail}^?\ell$ because $\text{int}! \# \text{bool}?\ell = \text{Fail}^?\ell$.

One of the subtle aspects of the Coercion Calculus is that coercion sequencing is associative and that all redexes are pairs of sequenced coercions. Thus, sometimes a sequence must be re-associated to reveal a redex. The last few lines of the definition of composition in Figure 4 deal with associativity by recursively invoking the composition operator. To ensure termination, we first check whether the coercion $c_1; (c_2; c_3)$ is in normal form. This check is easy to perform because there are only two such normal forms: $I^\ell!; (\iota \rightarrow \iota); I!$ and $I^\ell!; \langle C; C \rangle; I!$. The syntactic characterization of coercions in normal form is given at the bottom of Figure 4. The wrappers, written $\wr$, are coercions that may appear in a value. Also, the grammar for normal coercions relies on the auxiliary notion of a normal part, which excludes failure coercions.
\[
\text{\textbf{Syntax}}
\]

\[
\begin{align*}
\text{blame labels} & \quad \ell \in B & \text{member coercions} & \quad f \in F ::= T^{opt} (c \leftrightarrow c)^p T^{opt} \\
\text{optional labels} & \quad p \in B^{opt} ::= \epsilon, \ell & \text{injectables} & \quad I ::= B \mid T \rightarrow T \mid [\Delta; \Delta] \\
\text{member maps} & \quad C \in L \rightarrow F & \text{coercions} & \quad c \in C ::= \epsilon \mid I \mid I^\ell \mid \text{Fail}^\ell \\
\text{member maps} & \quad T^{opt} ::= \epsilon \mid T & \text{coercions} & \quad c : c \mid \epsilon \mid c \rightarrow c \mid [C; C]
\end{align*}
\]

\[
\begin{align*}
\vdash c : T & \Rightarrow T \\
\vdash \iota : T & \Rightarrow T & \vdash I! : I & \Rightarrow \star & \vdash I^\ell ! : \star & \Rightarrow I & \vdash \text{Fail}^\ell : T & \Rightarrow T_1 \\
\vdash c_1 : T_1 & \Rightarrow T_1 & \vdash c_2 : T_2 & \Rightarrow T_2 & \vdash c_1 : T_1 & \Rightarrow T_2 & \vdash c_2 : T_2 & \Rightarrow T_3 \\
\vdash c_1 \rightarrow c_2 : (T_1 \rightarrow T_2) & \Rightarrow (T_3 \rightarrow T_4) & \vdash c_1 \vdash c_2 : T_1 & \Rightarrow T_1 & \vdash c_1 \vdash c_2 : T_1 & \Rightarrow T_3 \\
\vdash C : \Delta_1 & \Rightarrow \Delta_3 & \vdash C_2 : \Delta_2 & \Rightarrow \Delta_4 & \vdash [C_1; C_2] : [\Delta_1; \Delta_2] & \Rightarrow [\Delta_3; \Delta_4]
\end{align*}
\]

\[
\begin{align*}
\vdash C_l : \Delta_1(l) & \Rightarrow \Delta_2(l) \quad \forall l \in \text{dom}(\Delta_1) \cap \text{dom}(\Delta_2) \\
\vdash C_l : \Delta_1(l) & \Rightarrow \epsilon \quad \forall l \in \text{dom}(\Delta_1) - \text{dom}(\Delta_2) \\
\vdash C_l : \epsilon & \Rightarrow \Delta_2(l) \quad \forall l \in \text{dom}(\Delta_2) - \text{dom}(\Delta_1) \\
\vdash C_l : \epsilon & \Rightarrow \epsilon \quad \forall l \in \text{dom}(C) - \text{dom}(\Delta_1) - \text{dom}(\Delta_2) \\
\vdash C : \Delta_1 & \Rightarrow \Delta_2
\end{align*}
\]

\[
\begin{align*}
\vdash f : T^{opt} & \Rightarrow T^{opt} \\
\vdash c_1 : T_2 & \Rightarrow T_1 & \vdash c_2 : T_1 & \Rightarrow T_2 & \vdash \epsilon (c_1 \leftarrow c_2)^! \epsilon : T_1 & \Rightarrow T_2
\end{align*}
\]

\[
\begin{align*}
\vdash c_1 : T_2 & \Rightarrow T_1 & \vdash c_2 : T_1 & \Rightarrow T_2 & \vdash \epsilon (c_1 \leftarrow c_2)^! T_2 : T_1 & \Rightarrow \epsilon
\end{align*}
\]

\[
\begin{align*}
\vdash c_1 : T_2 & \Rightarrow T_1 & \vdash c_2 : T_1 & \Rightarrow T_2 & \vdash \epsilon (c_1 \leftarrow c_2)^! T_2 & \Rightarrow \epsilon \\
\vdash T_1 (c_1 \leftarrow c_2)^! \epsilon & \Rightarrow \epsilon
\end{align*}
\]

\[
\begin{align*}
\langle T \Rightarrow T \rangle^! \\
\langle B \Rightarrow B \rangle^! & = \iota & \langle \star \Rightarrow \star \rangle^! & = \iota & \langle I \Rightarrow I \rangle^! & = I! & \langle \star \Rightarrow I \rangle^! & = I^\ell \\
\langle T_1 \Rightarrow T_2 \Rightarrow T_3 \Rightarrow T_4 \rangle^! & = \langle T_1 \Rightarrow T_2 \rangle^! \Rightarrow \langle T_2 \Rightarrow T_4 \rangle^! \\
\langle [\Delta_1; \Delta_2] \Rightarrow [\Delta_3; \Delta_4] \rangle^! & = \langle \Delta_1 \Rightarrow \Delta_3 \rangle^! \Rightarrow \langle \Delta_2 \Rightarrow \Delta_4 \rangle^!
\end{align*}
\]

\[
\begin{align*}
\langle \Delta \Rightarrow \Delta \rangle^! \\
\langle \Delta_1 \Rightarrow \Delta_2 \rangle^! (l) & = \begin{cases} 
\epsilon (\langle \Delta_2(l) \Rightarrow \Delta_3(l) \rangle^! \leftarrow \langle \Delta_1(l) \Rightarrow \Delta_2(l) \rangle^! \epsilon) & \text{if } l \in \text{dom}(\Delta_1) \cap \text{dom}(\Delta_2) \\
\epsilon (\iota \leftrightarrow \iota)^! \Delta_1(l) & \text{if } l \in \text{dom}(\Delta_1) - \text{dom}(\Delta_2) \\
\Delta_2(l) (\iota \leftrightarrow \iota)^! \epsilon & \text{if } l \in \text{dom}(\Delta_2) - \text{dom}(\Delta_1)
\end{cases}
\end{align*}
\]

\[
\overline{C}(l) = T^{opt}_{1} (c_2 \leftarrow c_1)^! T^{opt}_{2} \quad \text{where } C(l) = T^{opt}_{1} (c_1 \leftarrow c_2)^! T^{opt}_{2}
\]

Fig. 3. Coercions for mutable objects: syntax, type system, creation, and inversion.
\[ I_1 \downarrow I_2 \downarrow^\ell = \begin{cases} (I_1 \Rightarrow I_2)^\ell & \text{if } I_1 \sim I_2 \\ \text{Fail}^\ell & \text{otherwise} \end{cases} \]

\[ (c_1 \to c_2) \downarrow (c_3 \to c_4) = \begin{cases} \text{Fail}^\ell & \text{if } c_3 \downarrow c_1 = \text{Fail}^\ell \\ \text{Fail}^\ell & \text{if } c_2 \downarrow c_4 = \text{Fail}^\ell \\ (c_3 \downarrow c_1) \to (c_2 \downarrow c_4) & \text{otherwise} \end{cases} \]

\[ [C_1; C_2] \downarrow [C_3; C_4] = \begin{cases} \text{Fail}^\ell & \text{if } C_1 \downarrow C_3 = \text{Fail}^\ell \\ \text{Fail}^\ell & \text{if } C_2 \downarrow C_4 = \text{Fail}^\ell \\ (C_1 \downarrow C_3; C_2 \downarrow C_4) & \text{otherwise} \end{cases} \]

\[ \ell \downarrow c = c \downarrow \ell = c \quad \text{Fail}^\ell \downarrow c = \text{Fail}^\ell \quad I! \downarrow \text{Fail}^\ell = \text{Fail}^\ell \]

\[ (c_1; c_2) \downarrow c_3 = (c_1 \downarrow c_2) \downarrow c_3 \quad \text{if } c_1; (c_2; c_3) \text{ is not in normal form} \]

\[ c_1 \downarrow c_2 = c_1 \downarrow c_2 \quad \text{if } c_1; c_2 \text{ is in normal form} \]

\[ C \downarrow C \]

\[ C_1 \downarrow C_2 = \begin{cases} C_3 & \text{if } \forall l \in \text{dom}(C_1) \cap \text{dom}(C_2), C_1(l) \downarrow C_2(l) \neq \text{Fail}^\ell \\ \text{Fail}^\ell & \text{if } C_1(l) \downarrow C_2(l) = \text{Fail}^\ell \text{ for some } l \end{cases} \]

where \( C_3(l) = \begin{cases} C_1(l) \downarrow C_2(l) & l \in \text{dom}(C_1) \cap \text{dom}(C_2) \\ C_1(l) & l \in \text{dom}(C_1) - \text{dom}(C_2) \\ C_2(l) & l \in \text{dom}(C_2) - \text{dom}(C_1) \end{cases} \]

\[ f \downarrow f \]

\[ T_4^{\text{opt}}(c_1 \leftrightarrow c_2)^\ell \downarrow c(c_3 \leftrightarrow c_4)^\ell T_4^{\text{opt}} = \begin{cases} \text{Fail}^\ell & \text{if } c_3 \downarrow c_1 = \text{Fail}^\ell \\ \text{Fail}^\ell & \text{if } c_2 \downarrow c_4 = \text{Fail}^\ell \\ (c_3 \downarrow c_1) \leftrightarrow (c_2 \downarrow c_4) & \text{otherwise} \end{cases} \]

\[ T_4^{\text{opt}}(c_1 \leftrightarrow c_2)^\ell; T_2 \downarrow C_3(c_3 \leftrightarrow c_4)^\ell T_4^{\text{opt}} = \begin{cases} \text{Fail}^\ell & \text{if } c_3 = \text{Fail}^\ell \\ \text{Fail}^\ell & \text{if } c_2 = \text{Fail}^\ell \\ (c_3 \downarrow c_1) \leftrightarrow (c_2 \downarrow c_4) & \text{otherwise} \end{cases} \]

where \( c_3 = c_3 \downarrow (T_3 \Rightarrow T_2)^\ell \downarrow c_1 \)

and \( c_2 = c_2 \downarrow (T_2 \Rightarrow T_3)^\ell \downarrow c_4 \)

\[ p \downarrow p = p \]

\[ \ell \downarrow q = \ell \quad \epsilon \downarrow q = q \]

**Normal forms**

- **wrappers**
  \[ \tau ::= I! \mid \check{c} \to \check{c} \mid (\check{c} \to \check{c}) \mid [\check{C}; \check{C}] \mid [\check{C}; \check{C}]! \mid I! \]

- **normal parts**
  \[ \check{c} ::= \tau \mid I\ell^f \mid I\ell^s; \text{Fail}^f \mid I\ell^s; I! \mid I\ell^f \mid \check{c} \to \check{c} \mid I\ell^s; (\check{c} \to \check{c}) \mid I\ell^f \mid I\ell^s; \text{Fail}^f \mid I\ell^s; I! \mid [\check{C}; \check{C}] \mid [\check{C}; \check{C}]! \mid I\ell^f \mid I\ell^s; [\check{C}; \check{C}] \mid [\check{C}; \check{C}]! \mid I\ell^f \mid I\ell^s; [\check{C}; \check{C}] \mid [\check{C}; \check{C}]! \mid I\ell^f \mid I\ell^s; [\check{C}; \check{C}] \mid [\check{C}; \check{C}]! \mid I\ell^f \mid I\ell^s; [\check{C}; \check{C}] \mid [\check{C}; \check{C}]! \mid I\ell^f \mid I\ell^s; \text{Fail}^f \]

**normal coercions**

\[ \tilde{c} ::= \check{c} \mid \text{Fail}^f \]

---

**Fig. 4.** Composition of coercions in normal form.
4.2 Object Coercions

An object coercion has the form \([C_1; C_2]\) and converts an object from one object type \([\Delta_1; \Delta_2]\) to another \([\Delta_3; \Delta_4]\). The \(C_i\)'s are partial functions from labels to member coercions, which require some explanation. Let us consider a few examples that motivate the general form of member coercions. First, suppose we wish to cast member \(x\) from \(\text{int}\) to \(*\), that is, cast from \([x \mapsto \text{int}; \emptyset]\) to \([x \mapsto *; \emptyset]\). The object coercion might associate an injection \(\text{int}!\) with field \(x\).

However, because fields are mutable, we need two coercions, one for writing and one for reading, just like reference coercions [Herman et al., 2007]. So in this example we would associate both \(\text{int}?\) and \(\text{int}!\) with field \(x\).

\[
\triangleright [x \mapsto (\text{int}? \leftrightarrow \text{int}!); \emptyset] : [x \mapsto \text{int}; \emptyset] \Rightarrow [x \mapsto *; \emptyset]
\]

We also want coercions that add or remove members. We prefix a member’s coercions with a type \(T_1\) to indicate that we are adding a member of type \(T_1\), and suffix the coercions with \(T_2\) to indicate removal. Also, we may need a blame label to know which cast was responsible for the addition or removal. So the general form of a member coercion is \(T_{\text{opt}}1((c_1 \mapsto c_2)^pT_{\text{opt}}2)\), where \(T_1\) and \(T_2\) are optional types and \(p\) is an optional blame label. The following two examples demonstrate coercions that add and remove field \(x\).

\[
\triangleright [x \mapsto \epsilon((\epsilon \mapsto \epsilon)^c\epsilon; \emptyset); [\emptyset; \emptyset] \Rightarrow [x \mapsto \text{int}; \emptyset]
\]
\[
\triangleright [x \mapsto \epsilon((\epsilon \mapsto \epsilon)^c\epsilon; [\emptyset; \emptyset] \Rightarrow [\emptyset; \emptyset]
\]

These prefixes and suffixes must be kept distinct from the member’s coercions so that we do not mistake member addition or removal with coercions on the member’s value. For example, the first of the following coercions converts the \(x\) field from \(*\) to \(\text{int}\), whereas the second coercion adds field \(x\).

\[
\triangleright [x \mapsto \epsilon((\epsilon \mapsto \epsilon)^c\epsilon; [\emptyset; \emptyset] \Rightarrow [x \mapsto \text{int}; \emptyset]
\]
\[
\triangleright [x \mapsto \epsilon((\epsilon \mapsto \epsilon)^c\epsilon; [\emptyset; \emptyset] \Rightarrow [x \mapsto *; \emptyset]
\]

The type rule for object coercions, shown in Figure 3, takes four situations into account: the member is present in both the source and target type, only in one or the other, or in neither.

The composition of object coercions is defined in Figure 4. Similar to function coercions, if the composition of member coercions results in a failure, then the entire composition of object coercions results in failure. The composition of object coercions relies on the composition operator for member maps, \(C_1 \upharpoonright C_2\). If a member \(l\) is present in both \(C_1\) and \(C_2\), then we compose the two member coercions. Otherwise we just take the member coercion present in \(C_1\) or \(C_2\). The composition operator for member coercions, \(f_1 \upharpoonright f_2\), must handle two cases, depending on whether \(f_1\) performs member removal (and thus \(f_2\) performs member addition), or not. In the case of a removal and addition, we bridge the gap between \(T_2\) and \(T_3\) by creating the coercions \((T_2 \Rightarrow T_3)^{\ell_2}\) and \((T_3 \Rightarrow T_2)^{\ell_1}\).
expressions \[ e \in \mathcal{E} ::= k \mid \text{op}(e) \mid x \mid \lambda x : T. \mathcal{E} \mid e : \mathcal{E} \mid [F;M] \mid e.I := e \mid e \rightarrow I \mid e \rightarrow I := e \]

coercion expr. \[ \hat{e} ::= x \mid \iota \mid I \ell \mid I ! \mid \hat{e} \rightarrow \hat{e} \mid \hat{e} \mid [\hat{C};\hat{C}] \mid \text{dom} \hat{e} \mid \text{cod} \hat{e} \]

statements \[ s ::= x = e ; \mid x = e (e) ; \mid \text{return} e ; \mid \text{return} e (e) ; \mid \text{return} e (e) : \mathcal{E} ; \]

Fig. 5. Syntax of the intermediate language for guarded objects.

\[
\text{EvalMonad } A \equiv \text{Heap} \rightarrow (A \times \text{Heap}) + \{\text{blame}^{\ell}, \text{stuck}\}
\]

\[
x \leftarrow A; B \equiv \lambda \mu . \text{case } e \text{ of } \left\{ \begin{array}{ll}
\text{return } v & \equiv \lambda \mu . (v, \mu) \\
\text{blame } \ell & \Rightarrow \text{blame } \ell \\
\text{allocate}(v) & \equiv \lambda \mu . (a, \mu(a := v)) \text{ if } a \notin \text{dom}(\mu)
\end{array} \right.
\]

| stuck \Rightarrow \text{stuck} | \text{read}(a) & \equiv \lambda \mu . (\mu(a), \mu) |
| (v, \mu) \Rightarrow ([x := v]B)\mu | \text{write}(a, v) & \equiv \lambda \mu . (a, \mu(a := v)) |

Fig. 6. The evaluation monad (EvalMonad).

5 Guarded Objects

With the development of coercions for objects complete, we are ready to present a space-efficient abstract machine for guarded objects. This machine operates on the intermediate language defined in Figure 5. The type system of this intermediate language is straightforward, so its definition is relegated to the Appendix.

The intermediate language is in a kind of A-normal form [Flanagan et al., 1993]. We separate function calls from the rest of the expressions, making calls into statements and storing their results in temporary variables. The last statement form \( \text{return } e(e) : \hat{e} ; \) is for function calls that would be tail calls except for an outstanding cast. One of the goals of this abstract machine is to make sure that such function calls are just as space efficient as if they were actually tail calls. The translation from the \( \text{imp}_{\lambda}^\star \)-calculus to the intermediate language, which inserts casts and flattens function calls, only relies on standard techniques, so it is also given in the Appendix.

The definition of the abstract machine relies on an expression evaluation function \( \llbracket e \rrbracket_\rho \) (where \( \rho \) is an environment mapping variables to values) and a step function that executes one statement at a time, transitioning from one state to the next. Our machine must deal with abrupt termination due to cast errors and thread a global heap \( \mu \) through every computation, so to reduce syntactic clutter, we use the combination of a maybe monad and a state monad [Wadler, 1992]. This monad provides operators for sequencing computations, allocating on the heap, reading from the heap, and writing to the heap (Figure 6).

In Section 5.1 and 5.2 we describe expression evaluation and then the machine transitions, but first it is instructive to consider the values of the intermediate language, shown in Figure 7. The uncasted values (ranged over by \( \nu \)) include constants \( k \) and objects \([A_1;A_2]\), where \( A_1 \) gives the address for each field and \( A_2 \) gives the address for each method. An uncasted value may be wrapped in a
uncasted values \( \nu ::= k \mid [A; A] \mid \check{c} \)
values \( v \in V ::= \nu \mid \nu : c \mid (\lambda x y. \bar{z}, \rho, \bar{v}, b) \)
Booleans \( b ::= \text{true} \mid \text{false} \)

addresses \( a \in \mathbb{N} \)
addresses \( a \in \mathbb{N} \)
member addresses \( A \in \mathbb{L} \rightarrow \mathbb{N} \)
environments \( \rho \in \mathbb{X} \rightarrow V \)
heaps \( \mu \in \mathbb{N} \rightarrow V \)

Fig. 7. The definition of values for guarded objects.

single coercion, \( \nu : \tau \). A naive semantics would allow objects to be wrapped in arbitrarily many casts, just as was the case for functions in early work on gradual typing. But here we can compose coercions and therefore only need at most one coercion around an object. Normally, we would treat functions (closures) in the same way, but we can do better, as we explain in Section 5.2.

As mentioned in the introduction, the canonical forms lemma reveals whether overhead is needed in the form of run-time dispatching. As you can see below, a value of object type can either be an object or a casted object. We see in Section 5.1 how this affects expression evaluation.

Lemma 1 (Canonical Forms for Guarded Objects).

1. If \( \vdash v : B \), then \( v = k \) for some constant \( k \).
2. If \( \vdash v : \ast \), then \( v \) has the forms \( v = \nu : \tau \) or \( v = (\lambda x y. \bar{z}, \rho, \bar{v}, b) \), where the target type of \( \bar{c} \) is \( \ast \).
3. If \( \vdash v : [A_1; A_2] \), then either \( v = [A_1; A_2] \) or \( v = [A_1; A_2] : [C_1; C_2] \).
4. If \( \vdash v : T_1 \rightarrow T_2 \), then \( v = (\lambda xy. \bar{z}, \rho, \check{c}_1 \rightarrow \check{c}_2, b) \).

The most important piece of the abstract machine is the cast function, defined in Figure 8. This function takes a value and a coercion and returns a new value. The sixth equation is where we use coercion composition. The coercion \( c_2 \) is applied to a value already wrapped with \( c_1 \). We compose the two coercions and then call cast on the result. (The result of the composition may be the identity or failure coercion.) The first three equations of the definition are straightforward: identity coercions are discarded and failure coercions raise blame. The fourth equation wraps an uncasted value in a cast. The fifth equation wraps an object with a coercion, but it must also check that the object has all of the fields required by the coercion, and if not, raise blame. We defer discussion of casting closures to Section 5.2.

5.1 Expression Evaluation for Guarded Objects

We define the evaluation of expressions in Figure 9. The case for casts, \( e : \check{c} \) simply evaluates \( e \) and applies the cast function. The cases for member projection and assignment are interesting because they reveal the run-time overhead inherent in guarded objects. In each we must dispatch between two cases, depending on whether the receiving object is just an object \([A_1; A_2]\) or a casted object \([A_1; A_2] : [C_1; C_2]\). The uncasted case is straightforward, performing a read or write to the heap respectively for member projection or assignment. The case
\[
\begin{align*}
\text{cast}(\nu, i) &= \nu \\
\text{cast}(\nu, \text{Fail}) &= \text{blame} \ell \\
\text{cast}(\nu, c) &= \nu : c \\
\text{cast}(\nu : c) &= \text{if } c \neq [C_1; C_2] \\
\text{cast}([A_1, A_2], [C_1; C_2]) &= \begin{cases} 
\text{return } [A_1; A_2] : [C_1; C_2] & \text{if } \text{dom}(C_1) \subseteq \text{dom}(A_1) \\
\text{and } \text{dom}(C_2) \subseteq \text{dom}(A_2) \\
\text{blame } \ell & \text{if } C_i(l) = T_1(c_1 \mapsto c_2)^{T^\text{opt}_2} \\
& \text{and } l \notin \text{dom}(A_i), i \in \{1, 2\} 
\end{cases} \\
\text{cast}(\nu : c_1, c_2) &= \text{cast}(\nu, c_1 : c_2) \\
\text{cast}((\lambda x y, \overline{\pi}_1, \rho, c_2), c_2) &= \text{return } (\lambda x y, \overline{\pi}_2, \rho(f := (\lambda x y, \overline{\pi}_1, \rho, c_1, \text{false})), c_2, \text{true}) \\
& \text{where } \overline{\pi}_2 = \text{return } f(x : \text{dom}(y)) : \text{cod}(y); \\
\text{cast}((\lambda x y, \overline{\pi}, \rho, c_1, \text{true}), c_2) &= \begin{cases} 
\text{blame } \ell & \text{if } c_1 : c_2 = \text{Fail} \ell \\
\text{or } c_1 : c_2 = c ; \text{Fail} \ell & \text{otherwise} \\
\text{return } (\lambda x y, \overline{\pi}, \rho, c_1 : c_2, \text{true}) 
\end{cases}
\end{align*}
\]

Fig. 8. The \text{cast} function for guarded objects.

for a casted object is more subtle. For field projection, we read the value from the heap and then apply the \(c_2\) coercion. For method projection, we combine the \(c_2\) coercion with the inverse of the object coercion, \([C_1, C_2]\), to form a function coercion and apply that to the underlying closure. (The inversion operator is defined in Figure 3.) For field assignment, we apply the \(c_1\) coercion and for method updates, we combine the \(c_1\) coercion with the object coercion \([C_1, C_2]\) to form a function coercion that is applied to the closure.

5.2 Machine Transitions and Efficient Functions

We define the transitions of the abstract machine for guarded objects in Figure 10, in the form of the \text{step} function. The design of the \text{step} function makes function application efficient in both time and space. Regarding time, we do not want to dispatch on the kind of function values during a function call. Thus, we use a single value form for closures, shown in Figure 7, that contains a coercion and a flag to indicate whether the closure has ever been cast. When a closure is initially created, the coercion is an identity function coercion and the flag is set to false (see the case for lambda expressions in Figure 9). The closure is given a second parameter \(y\), which will be bound to its coercion. However, the body \(\overline{\pi}\) does not initially mention \(y\), so a closure that is never cast ignores its coercion. When a closure is cast for the first time, we generate a new closure that calls the original closure, but uses the \(y\) parameter to apply the coercion stored in the closure to the argument and return value. To support this, we extend the language of coercions with variables and \text{dom} and \text{cod} operators, forming the syntactic category \(\dot{c}\) (Figure 5).

Turning our attention to the second equation of \text{step} in Figure 10, we see the transition for function calls. In addition to passing the normal argument
Expression evaluation for guarded objects.

Fig. 9.
implications on the
5.3 The Dynamic Semantics of Guarded Objects

from a function call: we apply the coercion to the return value.

top of the stack. The coercions stored on the stack come into play upon return

program to an observable. In other words, an implementation of the

observe function is applied to the value.

can apply the same trick for member projection but functions provide a layer

parameter, so there is no overhead on the callee side. It would be nice if we

performing the cast from the caller to the callee, thereby removing caller-side

overhead. Furthermore, if the callee was never cast, then it ignores the coercion

v2, we also pass the function’s coercion c. Thus, we push the responsibility of

performing the cast from the caller to the callee, thereby removing caller-side

overhead. Furthermore, if the callee was never cast, then it ignores the coercion

parameter, so there is no overhead on the callee side. It would be nice if we
could apply the same trick for member projection but functions provide a layer of
direction that is not present in objects.

The design of the step function also addresses space efficiency for tail calls. On
a tail call (the fourth and fifth equations), we do not push a new procedure call
frame. To prevent the build-up of coercions, for the tail call and cast combination,
we take the outstanding coercion c2 and compose it with the coercion c3 at the

of indirection that is not present in objects.

Fig. 10. The step and eval functions for guarded objects.

v2, we also pass the function’s coercion c. Thus, we push the responsibility of

performing the cast from the caller to the callee, thereby removing caller-side

overhead. Furthermore, if the callee was never cast, then it ignores the coercion

parameter, so there is no overhead on the callee side. It would be nice if we

could apply the same trick for member projection but functions provide a layer

of direction that is not present in objects.

The design of the step function also addresses space efficiency for tail calls. On
a tail call (the fourth and fifth equations), we do not push a new procedure call
frame. To prevent the build-up of coercions, for the tail call and cast combination,
we take the outstanding coercion c2 and compose it with the coercion c3 at the

top of the stack. The coercions stored on the stack come into play upon return

from a function call: we apply the coercion to the return value.

5.3 The Dynamic Semantics of Guarded Objects

To conclude our treatment of guarded objects, Figure 11 defines the dynamic

semantics on the impGGrλ*-calculus as the eval function, which maps a source

program to an observable. In other words, an implementation of the impGGrλ*-calculus

is only required to produce the same observable as eval. The eval function

applies the cast insertion translation to convert e to a sequence of statements

π, and then applies the step* function. If the program terminates normally, the

observe function is applied to the value.
eval(e) = \begin{align*}
\text{observe}(v') & \quad \text{if } \text{step}^* \sigma_0 \mu_0 = ((\text{return } e; \rho, c \cdot e), \mu_1), \\
\text{blame } \ell & \quad \text{if } \text{step}^* \sigma_0 \mu_0 = \text{blame } \ell \\
\text{stuck} & \quad \text{if } \text{step}^* \sigma_0 \mu_0 = \text{stuck}
\end{align*}

where $\emptyset \vdash e : T$ and $\emptyset ; T \vdash e \sim \bar{s}$ and $\sigma_0 = (\bar{s}, \emptyset, \ell \cdot e)$ and $\mu_0 = \emptyset$

\[\text{observe}(k) = k \quad \text{observe}([A_1; A_2]) = \text{observe}([A_1; A_2] : [C_1; C_2]) = \text{object} \]
\[\text{observe}(\nu : I!) = \text{observe}(\nu : (\ell; I!)) = \text{dynamic} \]
\[\text{observe}((\lambda x. y. s, \rho, c, b)) = \text{function} \]

Fig. 11. The dynamic semantics of guarded objects, defined as the eval function.

6 Monotonic Objects

The primary design goal for monotonic objects is to remove the overhead that we saw in guarded objects regarding member projection and assignment. To accomplish this we need the following property: if an expression is of object type, then the resulting value must be an actual object and not a guarded object. However, we still want to allow convenient interaction between static and dynamic code, for example, an object of type $[x \mapsto \star; \emptyset]$ can be passed into a function expecting $[x \mapsto \text{int}; \emptyset]$. Our solution is to change the object during the cast: we check that the value stored in $x$ is an integer and then unbox it.

The overall invariant that we maintain is that the type $T_1$ of the value at address $a$ in the heap is equal to or less dynamic than (a naive subtype of) the type $T_2$ of any reference to $a$, so $T_1 \triangleleft_n T_2$. Thus, if we have an object $[x \mapsto a; \emptyset]$ of type $[x \mapsto \text{int}; \emptyset]$, then $T_1 \triangleleft \text{int}$ and so $T_1 = \text{int}$. So the value at address $a$ must be an integer and the implementation can read it with zero run-time overhead. We use the name monotonic objects because the type of a value in the heap is allowed to monotonically decrease with respect to naive subtyping.

The intermediate language for monotonic objects, shown in Figure 12, differs slightly from that of guarded objects. The abstract machine needs the static type of an object at object creation, so we record its type in the syntax for object creation. Also, we distinguish between field projections in which the type of the field is fully static ($\star$ does not appear in the type) or not. We write a fully static field projection as normal, $e.l$, whereas a non-static projection is annotated with the static type of the field, $e.lT$. We give the same treatment to field assignment and method access. The type system for this intermediate language is straightforward so its definition is in the Appendix, Figure 19.

Before describing expression evaluation, let us study the forms of values, defined in Figure 12. As before, we have constants, objects, and closures. However, we no longer have casted objects. We also make a change to the heap by storing
expressions \( e \in E ::= k \mid op(e) \mid x \mid \lambda x:T.\pi \mid e : \dot{c} \mid [F;M] as [\Delta;\dot{\Delta}] \mid e.I \mid e.I_T \mid e.l := e \mid e.I_T := e \mid e \rightarrow l \mid e \rightarrow L_T \mid e \rightarrow l := e \mid e \rightarrow L_T := e \)

statements \( s ::= \) same as for guarded objects

uncasted values \( \nu ::= k \mid [A;A] \)

values \( v \in V ::= \nu \mid \nu : I! \mid \langle \lambda x.y.\pi,\rho,\tau,\beta \rangle \)

labeled types \( L \in T^\ell ::= B^p \mid \star \mid [A;A]^p \mid L \rightarrow^p L \)

labeled type map \( A \in L \rightarrow T^\ell \times B^{opt} \)

heaps \( \mu \in N \rightarrow V \times T^\ell \)

Fig. 12. The intermediate language and values for monotonic objects.

a labeled type with each value. This type is the type of the value but annotated with blame labels that say which casts were responsible for each part of the type.

Lemma 2 (Canonical Forms for Monotonic Objects).

1. If \( \vdash v : B \), then \( v = k \) for some constant \( k \).
2. If \( \vdash v : \star \), then \( v \) has the form \( v = \nu : I! \) or \( v = \langle \lambda x.y.\pi,\rho,\tau,\beta \rangle \), where the target type of \( \pi \) is \( \star \).
3. If \( \vdash v : [\Delta_1;\Delta_2] \), then \( v = [A_1;A_2] \).
4. If \( \vdash v : T_1 \rightarrow T_2 \), then \( v = \langle \lambda x.y.\pi,\rho,\dot{c}_1 \rightarrow \dot{c}_2,\beta \rangle \).

Figure 13 defines expression evaluation for monotonic objects. The equations for constants, primitive operations, variables, and lambda expressions are the same as those for guarded objects. For object creation, the only change is that the type of each member is stored next to the value in the heap. The notation \( \lceil T \rceil \) creates a labeled type from a type by annotating each part of the type with \( \epsilon \). The next equation, for fully-static member projection, is beautiful. We simply read the member’s value from the heap. Similarly, the equation for fully-static member assignment just writes the new value to the heap.

The evaluation rules for non-static member access are subtle. The reference’s type for a member, \( T \), may be be more dynamic than the corresponding value in the heap. Thus, when projecting a member we must cast the value from its labeled type \( L \) to \( T \). We use the notation \( L \Rightarrow T \) for the operator that constructs a coercion from \( L \) to \( T \) and give its definition in Figure 13. (The notation \( \lfloor L \rfloor \) erases the labels from a labeled type to obtain a type, see Figure 22 in the Appendix.) Conversely, when assigning to a member, we must cast from \( T \) to \( L \).

Figure 13 also define the operator \( T \Rightarrow L \). The definitions of these two operators are mutually recursive because of the contravariance in function types. Because \( \lfloor L \rfloor <:_{n} T \), the operator \( L \Rightarrow T \) primarily produces identity coercions and injections, whereas \( T \Rightarrow L \) produces identities and projections. The auxiliary toplabel function just returns the top-most label in the type, and is defined in Figure 22 of the Appendix.

The last equation of Figure 13 looks the same as in guarded objects, but it is really quite different because the cast function behaves differently. The cast function must maintain the invariant that the values on the heap are less than
\[\begin{align*}
[[F; M]_{\Delta_1; \Delta_2}] & = v_1 \leftarrow [e_1] \rho; a_1 \leftarrow \text{alloc}((v_1, [\Delta_1; l_1])); \ldots \\
& \quad v_{n+1} \leftarrow [e_{n+1}] \rho; a_{n+1} \leftarrow \text{alloc}((v_{n+1}, [\Delta_1; \Delta_2] \Rightarrow \Delta_2(l_{n+1}))); \ldots \\
& \quad \text{return } [l_i = a_i \mid i = 1, \ldots, l_j = a_j \mid j = n+1, \ldots, m]
\end{align*}\]

where \(F = \{l_1 \mapsto e_1, \ldots, l_n \mapsto e_n\} \) and \(M = \{l_{n+1} \mapsto e_{n+1}, \ldots, l_m \mapsto e_m\}\)

\[\begin{align*}
[e; l] \rho & = [A_1; A_2] \leftarrow [e] \rho; (v, L) \leftarrow \text{read}(A_1(l)); \text{return } v \\
[e; \text{rt}] \rho & = [A_1; A_2] \leftarrow [e] \rho; (v, L) \leftarrow \text{read}(A_2(l)); \text{return } v \\
[e_1; l := e_2] \rho & = [A_1; A_2] \leftarrow [e_1] \rho; v_2 \leftarrow [e_2] \rho; \\
& \quad (\_; L) \leftarrow \text{read}(A_1(l)); \_ \leftarrow \text{write}(A_1(l), (v_2, L)); \text{return } [A_1; A_2] \\
[e_1; \text{rt}] \rho & = [A_1; A_2] \leftarrow [e_1] \rho; (v, L) \leftarrow \text{read}(A_2(l)); \text{return } [A_1; A_2] \\
[e_1; \text{rt}] \rho & = [A_1; A_2] \leftarrow [e_2] \rho; (v, L) \leftarrow \text{read}(A_1(l)); \\
& \quad v'_2 \leftarrow \text{cast}(v_2, T \Rightarrow L); \_ \leftarrow \text{write}(A_1(l), (v'_2, L)); \text{return } [A_1; A_2] \\
[e; e'] \rho & = v \leftarrow [e] \rho; \text{cast}(v, [e'] \rho)
\end{align*}\]

\[\begin{align*}
L \Rightarrow T = c
\end{align*}\]

\(\text{precondition: } [L] \prec \_; T, \text{ postcondition: } \vdash c : [L] \Rightarrow T\)

\[\begin{align*}
B^p & \Rightarrow B = \iota \quad \star \Rightarrow \star = \iota \quad \star \Rightarrow \lambda = [L]^{-1} \\
[A_1; A_2]^p & \Rightarrow [\Delta_3; \Delta_4] = [A_1 \Rightarrow \Delta_3; A_2 \Rightarrow \Delta_4] \\
L_1 \Rightarrow L_2 \Rightarrow T_3 \Rightarrow T_4 & = (T_3 \Rightarrow L_1) \Rightarrow (L_2 \Rightarrow T_4)
\end{align*}\]

\[\begin{align*}
A \Rightarrow A = A
\end{align*}\]

\(\text{precondition: } [A] \prec \_; A\)

\[\begin{align*}
(A \Rightarrow \Delta)(l) & = \begin{cases} 
(A(l) \Rightarrow \Delta(l), \epsilon) & \text{if } l \in \text{dom}(\Delta) \\
(A(l) \Rightarrow \Delta(l), A(l)) & \text{if } l \notin \text{dom}(\Delta)
\end{cases}
\end{align*}\]

\[\begin{align*}
T \Rightarrow L = c
\end{align*}\]

\(\text{precondition: } [L] \prec \_; T, \text{ postcondition: } \vdash c : T \Rightarrow [L]\)

\[\begin{align*}
B \Rightarrow B^p & = \iota \quad \star \Rightarrow \star = \iota \\
& \Rightarrow \lambda = [L]^\ell \quad \text{where } \ell = \text{toplabel}(L) \text{ and } L \neq \star \\
[A_1; A_2]^p & \Rightarrow [A_3; A_4] = [A_1 \Rightarrow A_3; A_3 \Rightarrow A_4] \\
T_1 \Rightarrow T_2 \Rightarrow L_3 \Rightarrow L_4 & = (L_3 \Rightarrow T_1) \Rightarrow (T_2 \Rightarrow L_4)
\end{align*}\]

\[\begin{align*}
\Delta \Rightarrow A = A
\end{align*}\]

\(\text{precondition: } [A] \prec \_; A\)

\[\begin{align*}
(\Delta \Rightarrow A)(l) & = \begin{cases} 
(\Delta(l) \Rightarrow A(l)_{\_1}, \epsilon) & \text{if } l \in \text{dom}(\Delta) \\
(\Delta(l) \Rightarrow A(l), A(l)) & \text{if } l \notin \text{dom}(\Delta)
\end{cases}
\end{align*}\]

Fig. 13. Expression evaluation for monotonic objects and auxiliary definitions.
or equally dynamic than any reference to them. In particular, we may need to unbox some values (or signal a cast error) to make the value’s type as static as the target type of the cast.

The notion of an object coercion is different with the monotonic approach because the act of casting an object does not build a new reference to the object, it changes the object’s representation in the heap. Figure 14 defines coercions for monotonic objects. The syntax of an object coercion now uses labeled type maps instead of member coercion maps: \([A_1; A_2]\) instead of \([C_1; C_2]\). The idea is that an object coercion changes an object so that its fields and methods are no more dynamic than the labeled types specified by \(A_1\) and \(A_2\), respectively.

With this change, the algorithm for composing two object coercions needs to be updated. We define what composition means for the labeled type maps and also for labeled types in Figure 14. The composition of labeled types computes the greatest lower bound with respect to naive subtyping (if it exists). A greatest lower bound exists if and only if the two types are consistent, that is, \(L_1 ; L_2 \neq \text{Fail}\) if and only if \([L_1] \sim [L_2]\). The composition operator can be viewed as an eager variant of the composition operator defined by Siek and Wadler [2010].

Figure 15 defines the \texttt{cast} function for monotonic objects. The equations are essentially the same as before except for objects. When casting an object, we invoke the auxiliary \texttt{Cast} function to coerce the object’s fields and methods and then return the object. For each member \(l_i\), the \texttt{Cast} function reads its value \(v_i\) and labeled type \(L_i\), then composes \(L_i\) with the labeled type in the cast, \(A(l_i)\) to get the new labeled type \(L'_i\) for the member. Then the member’s value \(v_i\) is cast to \(L'_i\), making use of the auxiliary operator \(L_i \sim L'_i\), defined in Figure 15. Finally, the new value is written back into the field.

The treatment of functions and function calls in monotonic objects is the same as for guarded objects. Thus, the \texttt{step} function for guarded objects can be adapted by simply replacing the expression and coercion evaluation functions and the \texttt{cast} function. The \texttt{eval} function can be similarly adapted with some minor changes to the \texttt{observe} function to handle the changes in the value forms. Thus, our formal description of monotonic objects is complete.

7  Related Work

Our work on monotonic objects was partly inspired by analogous work on type-state systems. Recall that typestates classify an object as being in one of several states, with each state potentially offering different fields and methods. Fähndrich and Leino [2003] introduce the notion of monotonic typestate, in which an object may proceed from less restrictive to more restrictive typestates, but not vice versa. Our work differs in that the monotonicity is enforced by a combination of static and dynamic checking (which is necessary for dealing with dynamically-typed regions of code) whereas with monotonic typestate, the monotonicity is enforced statically. Jagadeesan et al. [2009] adapt monotonic typestate to the \texttt{imp}\(\varsigma\)-calculus [Abadi and Cardelli, 1996].
coercions \( c \in C ::= e \mid I \mid I?^f \mid \text{Fail}^f \mid c \cdot c \mid c \rightarrow c \mid [A; A] \)

\[
\langle \Delta \Rightarrow \Delta \rangle^f =
\begin{cases}
(\lfloor \Delta_1(l) \rfloor^f \uparrow \lfloor \Delta_2(l) \rfloor^f, \epsilon) & \text{if } l \in \text{dom}(\Delta_1) \cap \text{dom}(\Delta_2) \\
(\lfloor \Delta_1(l) \rfloor^f, \epsilon) & \text{if } l \in \text{dom}(\Delta_1) - \text{dom}(\Delta_2) \\
(\lfloor \Delta_2(l) \rfloor^f, \epsilon) & \text{if } l \in \text{dom}(\Delta_2) - \text{dom}(\Delta_1)
\end{cases}
\]

\[c \uparrow c\]

\[
[A_1; A_2] \uparrow [A_3; A_4] =
\begin{cases}
\text{Fail}^f & \text{if } A_1 \uparrow A_3 = \text{Fail}^f \\
\text{Fail}^f & \text{if } A_2 \uparrow A_4 = \text{Fail}^f \\
[A_1 \uparrow A_3; A_2 \uparrow A_4] & \text{otherwise}
\end{cases}
\]

\[L_1 \uparrow L_2 = L_3 \text{ or } \text{Fail}^f\]

postcondition: \([L_3] <_{\text{in}} [L_1], [L_3] <_{\text{in}} [L_2]\)

\[
\begin{align*}
* \uparrow L & = L \\
L \uparrow * & = L \\
B^p \uparrow B^q & = B^{p,q}
\end{align*}
\]

\[
L_1 \rightarrow^p L_2 \uparrow L_3 \rightarrow^q L_4 =
\begin{cases}
\text{Fail}^f & \text{if } L_3 \uparrow L_1 = \text{Fail}^f \\
\text{Fail}^f & \text{if } L_2 \uparrow L_4 = \text{Fail}^f \\
(L_3 \uparrow L_1) \rightarrow^{p,q} (L_2 \uparrow L_4) & \text{otherwise}
\end{cases}
\]

\[
[A_1; A_2]^p \uparrow [A_3; A_4]^q =
\begin{cases}
\text{Fail}^f & \text{if } A_1 \uparrow A_3 = \text{Fail}^f \\
\text{Fail}^f & \text{if } A_2 \uparrow A_4 = \text{Fail}^f \\
[A_1 \uparrow A_3; A_2 \uparrow A_4]^{p,q} & \text{otherwise}
\end{cases}
\]

\[L_1 \uparrow L_2 = \text{Fail}^f \text{ if } \text{head}(L_1) \neq \text{head}(L_2) \text{ and } \text{toplabel}(L_2) = \ell
\]

\[\text{head}(B^p) = B \quad \text{head}([A_1; A_2]^p) = \text{object} \quad \text{head}(L_1 \rightarrow^p L_2) = \text{function}
\]

\[A_1 \uparrow A = A
\]

\[
A_1 \uparrow A_2 =
\begin{cases}
\text{Fail}^f & \text{if } A_1(l) \uparrow A_2(l) = \text{Fail}^f \text{ for some } l \\
A_3 & \text{otherwise}
\end{cases}
\]

where \(A_3(l) =
\begin{cases}
A_1(l) \uparrow A_2(l) & \text{if } l \in \text{dom}(A_1) \cap \text{dom}(A_2) \\
A_1(l) & \text{if } l \in \text{dom}(A_1) - \text{dom}(A_2) \\
A_2(l) & \text{if } l \in \text{dom}(A_2) - \text{dom}(A_1)
\end{cases}
\]

\[(L, p) \uparrow (L, p) = (L, p)
\]

\[
(L_1, p) \uparrow (L_2, q) =
\begin{cases}
\text{Fail}^f & = L_1 \uparrow L_2 = \text{Fail}^f \\
(L_1 \uparrow L_2, p \uparrow q) & \text{otherwise}
\end{cases}
\]

Fig. 14. Coercions for monotonic objects and composition of labeled types.
cast([A_1; A_2], [A_1; A_2]) = _ ← Cast(A_1, A_1); _ ← Cast(A_2, A_2); return [A_1; A_2]

Cast(A, A) =
\[
\begin{cases}
\text{blame } A(l_2) & \text{if } \exists \ l \in \text{dom}(A) \text{ and } l \notin \text{dom}(A) \\
(v_1, L_1) \leftarrow \text{read}(A(l_1)); \\
L'_1 \leftarrow L_1 \triangleright A(l_1); \\
v'_1 \leftarrow \text{cast}(v_1, c); \\
_1 \leftarrow \text{write}(A(l_1), (v'_1, L'_1));
\end{cases}
\]

\[L_1 \sim L_2 = c\] precondition: \(|L_2| <_n |L_1|\)

\(B^p \sim B^q = \iota \quad \star \sim \star = \iota \quad \star \sim L = [L]^p\) (where \(\ell = \text{toplabel}(L)\) and \(L \neq \star\))

\([A_1; A_2]^p \sim [A_3; A_4]^q = [A_3; A_4] \quad L_1 \to^p L_2 \sim L_3 \to^q L_4 = (L_3 \sim L_1) \to (L_2 \sim L_4)\)

Fig. 15. The cast function and auxiliary definitions.

The work on gradual typestate by Wolff et al. [2011] is closely related to ours in that typestate checking occurs both statically and dynamically. However, their system does not rely on monotonicity but instead uses reference counting to make sure that an object is not changed to a different typestate when there are outstanding references to it that rely on the old typestate. We decided against such an approach because of concerns over the predictability of cast errors and the run-time cost of reference counting.

The work on Typed Scheme [Tobin-Hochstadt and Felleisen, 2006, 2008, Takikawa et al., 2012] falls into the same category as our guarded objects, in that values are wrapped with guards to ensure they behave as expected. However, the design for Typed Scheme does not address space efficiency.

8 Conclusion

The design and implementation of gradually-typed languages with mutable objects is challenging. It is difficult to ensure type safety while also enabling fast execution of statically-typed code. This paper presents the first design, named monotonic objects, that does both. This design is especially suitable in situations that require high performance in parts of an application and where the intention of the programmer is to eventually migrate the entire program to static typing.

This paper also develops an alternative approach, guarded objects, that is the natural extension of earlier coercion-based implementations of gradual typing. Guarded objects induce some overhead in statically-typed code, but they are less restrictive than monotonic objects and are therefore more suitable in situations where the bulk of the program is written in a highly dynamic style.
Bibliography


Appendix

This appendix supplies additional definitions concerning the semantics of guarded and monotonic objects.

Guarded objects

We describe the type system of guarded objects and their coercions in Figure 16. This figure extends the type system of the \texttt{imp}_{\mathcal{T}}\lambda^*\text{-calculus}, as shown in Figure 2, and the type system of object coercions, as shown in Figure 3. It also defines the type system for statements in the intermediate language for guarded objects.

The coercion inserting translation from the \texttt{imp}_{\mathcal{T}}\lambda^*\text{-calculus} to the intermediate language for guarded objects is shown in Figures 17 and 18.

Monotonic objects

Figure 19 defines the type system for the intermediate language of monotonic objects. It extends the type system of guarded objects, in Figure 16, and likewise inherits from Figures 2 and 3.

Figure 20 extends and overrides Figures 17 and 18 to define the coercion inserting translation from the \texttt{imp}_{\mathcal{T}}\lambda^*\text{-calculus} to the intermediate language for monotonic objects. Figure 21 shows the judgment that determines whether or not a type is fully-static (lacking \texttt{*}), and therefore whether or not member accesses and writes to a field or method can be direct.

Figure 22 specifies several straightforward utility functions used during the casting process of monotonic objects. The \texttt{toplabel} function returns the outermost blame label (if one exists) of a labeled type. The $[L]$ function erases the labels from a labeled type to acquire a normal type, and the $[T]^p$ function applies a blame label to a normal type to create a labeled type.
Fig. 16. The type system for the intermediate language of guarded objects.
Fig. 17. Coercion insertion and conversion to ANF for guarded objects, part 1.
\[ \Gamma \vdash \llbracket e \rrbracket \sim \llbracket \bar{\alpha} \rrbracket : T \]

\[
\begin{align*}
e \neq e_1, e_2 & \quad \Gamma \vdash \llbracket e \rrbracket \sim \llbracket \bar{\alpha}' \rrbracket : T_1 \\
\Gamma \vdash \llbracket e \rrbracket \sim \llbracket \bar{\alpha} \rrbracket \text{ return } e' : T_1 \\
\Gamma \vdash \llbracket e_1 \rrbracket \sim \llbracket \bar{\alpha}_1 \rrbracket, \llbracket e_1' \rrbracket : T_1 \Rightarrow T_2 \\
\Gamma \vdash \llbracket e_2 \rrbracket \sim \llbracket \bar{\alpha}_2 \rrbracket, \llbracket e_2' \rrbracket : T_3 \quad \ell \text{ fresh} \quad T_1 \sim T_3
\end{align*}\]

\[
\begin{align*}
\Gamma \vdash \llbracket e_1, e_2 \rrbracket \sim \llbracket \bar{\alpha}_1, \bar{\alpha}_2 \rrbracket \text{ return } (e' : (T_3 \Rightarrow T_1)') : T_2 \\
\Gamma \vdash \llbracket e_1 \rrbracket \sim \llbracket \bar{\alpha}_1 \rrbracket, \llbracket e_1' \rrbracket : * \\
\Gamma \vdash \llbracket e_2 \rrbracket \sim \llbracket \bar{\alpha}_2 \rrbracket, \llbracket e_2' \rrbracket : T_1 \quad \ell_1 \text{ fresh}
\end{align*}\]

\[
\begin{align*}
\Gamma \vdash \llbracket e_1, e_2 \rrbracket \sim \llbracket \bar{\alpha}_1, \bar{\alpha}_2 \rrbracket \text{ return } (e' : (T_2 \Rightarrow T_1)') : T_1 \\
\Gamma \vdash \llbracket e \rrbracket \sim \llbracket \bar{\alpha}_1(e_2) \rrbracket : T_2 \quad \ell \text{ fresh}
\end{align*}\]

\[
\begin{align*}
\Gamma \vdash \llbracket e : T_1 \rrbracket \sim \llbracket \bar{\alpha}_1 \rrbracket \text{ return } e_1(e_2) : (T_2 \Rightarrow T_1)' : T_1 \\
\Gamma \vdash \llbracket e \rrbracket \sim \llbracket \bar{\alpha}_1 \rrbracket, \llbracket e_1(e_2) \rrbracket : T_2 \quad \ell \text{ fresh}
\end{align*}\]

\[
\begin{align*}
\Gamma \vdash \llbracket e : T_1 \rrbracket \sim \llbracket \bar{\alpha}_1 \rrbracket \text{ return } e_1(e_2) : e \mid (T_2 \Rightarrow T_1)' : T_1
\end{align*}\]

Fig. 18. Coercion insertion and conversion to ANF for guarded objects, part 2.

\[ \Gamma \vdash e : T \]

\[
\begin{align*}
\text{dom}(F) \cap \text{dom}(M) &= \emptyset \\
\forall l \in \text{dom}(\Delta_1), \Gamma \vdash F(l) : \Delta_1(l) \\
\forall l \in \text{dom}(\Delta_2), \Gamma \vdash M(l) : [\Delta_1 ; \Delta_2] \Rightarrow \Delta_2(l)
\end{align*}\]

\[
\begin{align*}
\Gamma \vdash e : [\Delta_1 ; \Delta_2] \\
l \in \text{dom}(\Delta_1) \quad T = \Delta_1(l) \\
\Gamma \vdash e, l_F : T
\end{align*}\]

\[
\begin{align*}
\Gamma \vdash e_1 : [\Delta_1 ; \Delta_2] \\
l \in \text{dom}(\Delta_1) \quad T = \Delta_1(l) \\
\Gamma \vdash e_1, l_F := e_2 : [\Delta_1 ; \Delta_2]
\end{align*}\]

\[
\begin{align*}
\Gamma \vdash \hat{e} : T
\end{align*}\]

\[
\begin{align*}
\forall l \in \text{dom}(\Delta_1). \Delta_1(l) <_n [A_1(l)] \land \Delta_2(l) <_n [A_2(l)] \\
\forall l \in \text{dom}(\Delta_2). \Delta_2(l) <_n [A_1(l)] \land \Delta_4(l) <_n [A_2(l)]
\end{align*}\]

\[
\begin{align*}
\Gamma \vdash [\Delta_1 ; \Delta_2] : [\Delta_1 ; \Delta_2] \Rightarrow [\Delta_3 ; \Delta_4]
\end{align*}\]

Fig. 19. The type system of the intermediate language for monotonic objects.
Fig. 20. Modifications to coercion insertion for monotonic objects.
\[ \vdash \text{static}(T) \]

\[ \vdash \text{static}(T_1) \quad \vdash \text{static}(T_2) \]

\[ \forall l \in \text{dom}(\Delta_1). \vdash \text{static}(\Delta_1(l)) \quad \forall l \in \text{dom}(\Delta_2). \vdash \text{static}(\Delta_2(l)) \]

\[ \vdash \text{static}(T_1 \to T_2) \]

\[ \forall l \in \text{dom}(\Delta_1 \cdot \Delta_2). \vdash \text{static}(\Delta_1 \cdot \Delta_2(l)) \]

**Fig. 21.** Fully-static types

\[ \text{toplabel}(L) = p \]

\[ \text{toplabel}(B^p) = p \]

\[ \text{toplabel}([A_1; A_2]^p) = p \]

\[ \text{toplabel}(L_1 \to^p L_2) = p \]

\[ [L] = T \]

\[ [\ast] = \ast \]

\[ [B^p] = B \]

\[ [[A_1; A_2]^p] = [[A_1]; [A_2]] \]

\[ [L_1 \to^p L_2] = [L_1] \to^p [L_2] \]

\[ [A](l) = [A(l)] \]

\[ [T]^p = L \]

\[ [B]^p = B^p \]

\[ [\ast]^p = \ast \]

\[ [T_1 \to T_2]^p = [T_1]^p \to^p [T_2]^p \]

\[ [\Delta_1; \Delta_2]^p = [[\Delta_1]^p; [\Delta_2]^p]^p \]

\[ [\Delta]^p(l) = [\Delta(l)]^p \]

**Fig. 22.** Some operations on labeled types.