Gradual Algebraic Datatypes

STEFAN MALEWSKI, University of Chile, Chile
MICHAEL GREENBERG, Pomona College, USA
ÉRIC TANTER, University of Chile, Chile

Dynamically-typed languages offer easy interaction with ad hoc data such as JSON and S-expressions; statically-typed languages offer powerful tools for working with structured data, notably *algebraic datatypes*, which are a core feature of typed languages both functional (OCaml, Haskell) and otherwise (Scala, Rust). Gradual typing aims to reconcile dynamic and static typing smoothly. The gradual typing literature has extensively focused on the computational aspect of types, such as type safety, effects, noninterference, or parametricity, but the application of graduality to data structuring mechanisms has been much less explored. While row polymorphism and set-theoretic types have been studied in the context of gradual typing, algebraic datatypes in particular have not, which is surprising considering their wide use in practice. We develop, formalize, and prototype a novel approach to gradually structured data with algebraic datatypes. Gradually structured data bridges the gap between traditional algebraic datatypes and flexible data management mechanisms such as tagged data in dynamic languages, or polymorphic variants in OCaml. We (a) illustrate the key ideas of gradual algebraic datatypes through the evolution of a small server application from dynamic to progressively more static checking, (b) formalize a core functional language with gradually structured data, and (c) establish its metatheory, including the gradual guarantees.

1 INTRODUCTION

Most matters in programming can either be entirely resolved at runtime, or count on the help of static type checking to enforce basic guarantees. Data structuring mechanisms are no exception. Dynamically-typed programming languages like JavaScript, Scheme, and Python use ad-hoc, semi-structured datatypes to support a prototype-based approach to development. The archetypal example is S-expressions, thanks to which one can simply represent data as a list with a 'tag' symbol at its head indicating the kind of data, followed by arbitrary elements. As prototypes grow, though, it can be challenging to safely evolve semi-structured datatypes. It is very easy to miss an uncommon path and forget to add support for new parts of the datatype or update support for changed ones. Statically-typed languages excel at this kind of maintenance, pointing programmers directly to the changed cases. Indeed, static datatype definitions are more rigid but bring stronger guarantees, such as ensuring that all possible alternatives of a given datatype have been considered. In any case, statically-typed languages must also deal with semi-structured data, which can be found all over the web [Buneman 1997], in formats such as JSON, S-XML, or indeed XML itself. Dealing with semi-structured data then requires falling back on alternatives such as string manipulations or generic structures with reflection or runtime type coercions in order to access specific attributes. Notably,

Authors' addresses: Stefan Malewski, Computer Science Department (DCC), University of Chile, Chile; Michael Greenberg, Pomona College, USA; Éric Tanter, Computer Science Department (DCC), University of Chile, Chile.

the OCaml programming language supports *polymorphic variants* [Garrigue 1998] as a more expressive and statically safe mechanism. Polymorphic variants are however completely detached syntactically from the standard way of structuring data, namely *algebraic datatypes*. For instance, a case expression cannot mix both algebraic datatypes and polymorphic variants. In addition, the type system machinery is also widely different. This strict separation of worlds complicates evolving code from one mechanism to the other.

This tension in data structuring mechanisms is reminiscent of the general tension between static and dynamic type checking, which has attracted a large body of work, in particular in gradual typing [Siek and Taha 2006]. Gradual typing allows for the smooth integration of static and dynamic typing, supporting both extremes as well as the continuum between them. Gradual typing has been explored for a variety of language features, such as *subtyping* [Garcia et al. 2016; Siek and Taha 2007; Takikawa et al. 2012], *references* [Herman et al. 2010; Siek et al. 2015b; Toro and Tanter 2020], *effects* [Bañados Schwerter et al. 2014, 2016], *ownership* [Sergey and Clarke 2012], *typestate* [Garcia et al. 2014; Wolff et al. 2011], *session types* [Igarashi et al. 2017b], *refinements* [Lehmann and Tanter 2017], *type inference* [Garcia and Cimini 2015; Vazou et al. 2018] *parametricity* [Ahmed et al. 2011, 2017; Igarashi et al. 2017a; New et al. 2020; Toro et al. 2019], etc.

However, the interaction of gradual typing with data structuring mechanisms has been rather scarce. Notable exceptions are *row polymorphism* [Garcia et al. 2016; Sekiyama and Igarashi 2020], *set-theoretic types* [Castagna and Lanvin 2017; Castagna et al. 2019] and *union types* [Siek and Tobin-Hochstadt 2016; Toro and Tanter 2017]. For instance, Sekiyama and Igarashi [2020] devise a language with row polymorphism and gradual records. In row types, every variant is its own type, similar to Typed Racket [Tobin-Hochstadt and Felleisen 2008]. Row types are very flexible, allowing for unknown variants, but they do not directly support the more conventional definitions of nominal datatypes with a static set of variants.

This work explores the application of gradual typing to nominal algebraic datatypes as found in many popular languages such as OCaml, Haskell, Scala, Elm, Rust, etc. We consider both closed and open datatypes as found in e.g. Scala. We show that gradualizing such a language yields a novel, expressive approach to gradually structured data: a simple type system allows for imprecise static information while keeping a single namespace for all constructors, whether or not they are statically declared. Gradual languages trade off between checking invariants statically and at runtime. When adding algebraic datatypes to a language, the guarantees we are after are that all pattern matches are complete, all constructors are statically known, and every constructor belongs to at most one datatype. As we “go gradual”, these properties are no longer statically guaranteed. Care must be taken to ensure that runtime checks allow moving programs seamlessly across the static/dynamic spectrum; that is, the resulting language must not only satisfy type soundness, but the gradual guarantees [Siek et al. 2015a] as well.

Contributions. This article presents the following contributions:

- We illustrate the use of gradually structured data in a novel language, GSD (for Gradually Structured Data), through the evolution of a web API of a simple arithmetic interpreter (Section 2).

- In addition to the standard *unknown type*, we identify the need for two novel gradual types related to datatypes: the *unknown datatype* and the *unknown open datatype*.
- We present a core statically-typed language with extensible, nominal algebraic datatypes called λ_{D} (Section 3) and develop its corresponding gradual language $\lambda_{\mathrm{D?}}$ (Section 4).
- We develop the runtime semantics of $\lambda_{\mathrm{D?}}$ using an evidence-based intermediate language, following the simplified approach to AGT proposed by Toro et al. [2019].
- We prove that $\lambda_{\mathrm{D?}}$ satisfies the expected criteria of [Siek et al. 2015a] for gradually-typed languages (Section 4.4).
- We derive $\lambda_{\mathrm{D?}}$ from λ_{D} following the Abstracting Gradual Typing (AGT) methodology [Garcia et al. 2016], thereby providing another case in favor of this systematic approach to gradual language design. Using AGT ensures that the semantics of $\lambda_{\mathrm{D?}}$ is obtained systematically from that of λ_{D}, and that the gradual guarantees are easy to uphold.
- We provide an implementation of GSD, a practical language built on top of $\lambda_{\mathrm{D?}}$, together with a number of illustrative examples, available at https://pleiad.cl/gsd .

Section 5 compares GSD and related languages, Section 6 discusses other related work, and Section 7 concludes. We omit some (parts of) definitions; full definitions and proofs are in supplementary material.

2 GRADUALLY STRUCTURED DATA IN ACTION

We now illustrate the use of gradually structured data as supported in our prototype language GSD (for Gradually Structured Data). We walk through a four-step development scenario for the web API of an arithmetic interpreter in GSD. The first version is fully dynamic, the fourth is almost completely statically typed. Along the way we highlight the most important features related to gradually structured data.

The Basic Arithmetic Server (BAS) is a simple web API for doing arithmetic calculations. Like most contemporary web APIs, it communicates by sending and receiving JSON messages. Also like most contemporary web APIs, we implement it with a prototype-based approach: we build a core of functionality in the server, but the server evolves as its clients do. There is no fixed "protocol" up front, only conventions. As the API becomes stable, the protocol becomes more fixed. Practically speaking, this means the development starts out mostly untyped, and that developers only add static types as parts of the development stabilize.

Version 1. The first version of BAS supports addition and subtraction (Figure 1). The server itself is the `serve` function (line 1), which takes a JSON request and returns a result (or error) as a JSON string, using `handleRequest` (line 2) to actually perform the request.

To handle a request (lines 3-5), first the API key is checked with `withValidKey` and then pattern matching determines what the request actually is. In this first version, BAS only supports addition (`Plus`) and subtraction (`Minus`). All other operations fail with a string error message (`Fail`). The definition of `withValidKey` (line 6) conveniently uses a direct field access (instead of pattern matching) to extract the `key` attribute of

```
serve jsonReq = toJSON (handleRequest (fromJSON jsonReq))
handleRequest request = withValidKey request (\r ⇒ match r with
                                              Plus key x y  ⇒ Success {r = (x + y)}
                                              Minus key x y ⇒ Success {r = (x - y)}
                                              _ ⇒ Fail {r =  "Error: unknown command"})
withValidKey r action = if isValidKey r.key -- key validation details omitted
                        then action r else Fail {r = "Error: invalid key"}
```

Fig. 1. BAS Version 1: dynamically-typed, supporting addition, subtraction, and API keys

the given request; we omit the definition of `isValidKey`, which could be as complicated as a database call or as simple as a checksum.

The `fromJSON` function parses the JSON input to *constructed data*. Constructed data is similar to an S-expression: it consists of a constructor name as a tag, followed by zero or more arguments. In GSD, `fromJSON` expects its input to be an object with just one field. The field's name becomes the constructor (i.e. tag) and the field's value the constructor arguments. For example:

```
> fromJSON '{"Plus": {"key":10, "x":1, "y":2}}'
Plus { key = 10, x = 1, y = 2 }
> fromJSON '{"Sqrt": {"key":10, "x": {"Frac": {"numerator": 11, "denominator": 12}}}}'
Sqrt { key = 10, x = Frac { numerator = 11, denominator = 12 } }
```

To seamlessly support this, constructors have labeled parameters; arguments with non-primitive types use the field name as the outermost tag (e.g., `Frac`). Keeping fields and tags ensures no information is lost parsing JSON strings into data. Dually, `toJSON` serializes a data value to a JSON string. Both of these functions are built-ins of GSD, because constructor name generation is not first class.

The dynamically-typed BAS Version 1 handles API requests appropriately (assuming `10` is a valid key):

```
> serve '{"Plus": {"key":10, "x":1, "y":2}}'
'{"Success": 3}'
> serve '{"Times": {"key":10, "x":1, "y":2}}'
'{"Fail": "Error: unknown command"}'
```

BAS Version 1 uses unstructured data: there are no declared datatypes. If the programmer were to type `Foil` instead of `Fail` in the error case, there would be no static error, and `toJSON` would send a confusing message to a client.

Adding types. GSD is gradually typed. Unannotated binders are considered to have the unknown type ?. As usual, ? is the least precise type, and is consistent with any other type. Recall that in gradual typing, type precision ($\sqsubseteq$) characterizes the amount of static information conveyed by a gradual type. For instance, $\mathsf{Int} \to \mathsf{Int} \sqsubseteq \mathsf{Int} \to ? \sqsubseteq ? \to ? \sqsubseteq ?$. Two types are *consistent* with each other there is a way to fill in their ? parts to reach an equal type, e.g. $\mathsf{Int} \sim ?$ but $\mathsf{Int} \nsim \mathsf{Int} \to ?$. Concretely, this means that a variable of type ? can be used in any position regardless of its expected type, and any value can flow to such a variable as well.

But adding datatypes to a language that only supports ? would only allow for the definition of constructors with unknown parameter types, it would not support *unclassified data*. Furthermore, having only the unknown type at hand would make it hard to precisely characterize when pattern matching can proceed: at runtime, the discriminee must be some constructed data, no matter from which datatype. In fact, this data may be from a statically-defined algebraic datatype, or unclassified.

To precisely characterize the minimal shape of values that can be pattern-matched, GSD introduces a new gradual type, the unknown datatype $?_{\mathsf{D}}$. This type can be understood as the "ground type" of data values (which can be eliminated by pattern matching), just like $? \rightarrow ?$ is the ground type of functions (which can be eliminated by function application). Therefore, pattern matching is only well-typed if the discriminee has a type consistent with $?_{\mathsf{D}}$: pattern matching on an expression of type $? \rightarrow ?$ or `Int` is a static type error. Likewise, it is a static type error to try to use a $?_{\mathsf{D}}$-typed expression in another elimination form, such as a function application, or primitive operation like addition.

Additionally, GSD must assign a type to unclassified data. Using $?_{\mathsf{D}}$ would be too imprecise to be satisfactory. Indeed, it would allow unclassified data to be optimistically considered as part of *closed* datatypes, which one expects to remain closed. When statically-typed code matches on an expression of a datatype with, say, two variants, the pattern match is expected to be exhaustive and not let unclassified data through. Therefore, GSD introduces another gradual type, the unknown open datatype $?_{\mathsf{O}}$, which is more precise than the unknown datatype $?_{\mathsf{D}}$, and is only consistent with *open* datatypes. Unclassified data has type $?_{\mathsf{O}}$ since it is considered as possibly inhabiting any *open* datatype. In summary, in GSD we have $?_{\mathsf{O}} \sqsubseteq ?_{\mathsf{D}} \sqsubseteq ?$. Given a closed datatype D, we have $D \sqsubseteq ?_{\mathsf{D}}$ but $D \nsim ?_{\mathsf{O}}$.

In Figure 1, matching on the `r` variable on line 2 is well-typed: `r` has type ?, which is consistent with $?_{\mathsf{D}}$. At runtime, when unclassified data (of type $?_{\mathsf{O}}$) flows into `r`, the matching reduces successfully because $?_{\mathsf{O}}$ is more precise than $?_{\mathsf{D}}$. Importantly, if `fromJSON` somehow returned a number instead of a data value, the match would fail at runtime. In fact, `fromJSON` has the type `String` $\rightarrow ?_{\mathsf{D}}$ in GSD, meaning that it necessarily returns data values, which can either be unclassified, or from statically-declared datatypes. This also means that clearly invalid uses of `fromJSON` such as `(fromJSON s) + 1` can be statically rejected. Finally, because `handleRequest` expects values of type ?, the definition of `serve` is well-typed because $?_{\mathsf{D}}$ is more precise than—and consistent with—?.

Version 2. In BAS Version 2, the developers start to firm up some of their definitions as the application evolves (Figure 2). First, the `Response` datatype is explicitly declared as a *closed* datatype with two alternatives, one for `Success` and one for `Failure`. A *closed* datatype is like a standard OCaml or Haskell algebraic datatype, in that all constructors are specified in place. Contrastingly, the developer remains uncertain of which requests and errors might occur, so the `Error` and `Request` datatypes are declared *open*. An *open* datatype admits the declaration of new variants after the datatype declaration.[1] With these datatype

[1]Object-oriented class extensibility typically defaults to open, with keywords like `final` indicating closed types. Scala supports algebraic datatypes with *case classes*. They are open by default (i.e. new variants can be added in other files/modules), and Scala

```
data Response = Success {x : ?} | Fail {msg : String}
open data Error
open data Request
serve : String → String                                   -- unchanged definitions
withValidKey : Request → (Request → Response) → Response  -- omitted
handleRequest : Request → Response
handleRequest request = withValidKey request (\r : Request ⇒  match r with
                                               Plus key x y  ⇒ Success (x + y)
                                               Minus key x y ⇒ Success (x - y)
                                               _             ⇒ Fail (msg CommandError))
msg : Error → String
msg err = match err with CommandError    ⇒ "unknown command"
                         InvalidKeyError ⇒ "invalid key"
                         _               ⇒ "unknown error"
```

Fig. 2. BAS Version 2: explicit response structure, but undetermined request and error structure

declarations in hand, the programmer can now give explicit types to most functions, such as `handleRequest`, of type `Request → Response`. BAS Version 2 has narrowed down many of its representations, even as it leaves some open—like `Error` and `Request`. Errors are no longer mere strings, but structured data—the new `msg` function pattern matches on them to produce string-based error messages. In GSD, open datatypes can be optimistically inhabited by *any* constructed data, so it is valid to match a value of type `Error` with patterns of some unanticipated constructor names. The programmer accounts for this with a catch-all _ case. Programmers familiar with polymorphic variants in OCaml will recognize this intermediate structuring of data. The difference at this point is that in GSD there is no polymorphic structural type inference as in OCaml, just simple gradual types. Many errors can still happen at runtime in GSD! For example, `handleRequest (Plus {key=1, x=False, y=7})` fails at runtime when `handleRequest` tries to evaluate `False + 7` and `handleRequest (Plus {key=1, x=False, y=7, z=0})` produces a `CommandError`, because there is no pattern matching a `Plus` constructor with four arguments.

Version 3. BAS Version 3 makes the `Error` and `Request` datatypes more static (Figure 3). For instance, `Plus` and `Minus` have been promoted to official constructors of the `Request` datatype. These explicit declarations fix the internal structure of these declared constructors: now `Plus {key=1, x=False, y=7, z=0}` is a *static* type error, because the constructor is used with the wrong number of arguments. Also, because `Plus` is now a declared constructor of the `Request` datatype, `Plus` is no longer a possible constructor of another open datatype: passing `Plus k x y` to `errorMsg` would yield a *static* type error.

Note that the types of the `x` and `y` fields of both `Plus` and `Minus` have type ? at this stage. So not every such error is static: `handleRequest (Plus {key=1, x=False, y=7})` fails at runtime when trying to add `False` and `7`.

provides the keyword `sealed` for closed case classes, requiring all variants to be locally defined, as in a typical functional programming language like Haskell or OCaml.

```
data Response = Success { x : ? } | Fail { msg : String }
open data Error = CommandError | InvalidKeyError        -- Declaring possible constructors
open data Request = Plus  { key : Int, x : ?, y : ? }  --
                  | Minus { key : Int, x : ?, y : ? }  --
handleRequest : Request → Response
handleRequest request =  withValidKey request (\r : Request ⇒ match r with
                                                Plus key x y  ⇒ Success (x + y)
                                                Minus key x y ⇒ Success (x - y)
                                                Not key x     ⇒ Success (not x)
                                                _             ⇒ Fail (msg CommandError))
```

Fig. 3. BAS Version 3: partially-specified datatypes (unchanged definitions omitted)

```
data Response = Success {x : Data} | Fail {msg : String}
data Data = N {x : Int} | B {x : Bool}
data Error = InvalidKeyError          -- | CommandError is no longer possible
data Request = Plus {key : Int, x : Int, y : Int}
             | Minus {key : Int, x : Int, y : Int}
             | Not {key : Int, x : Bool}
handleRequest : Request → Response
handleRequest request = withValidKey request (\r : Request ⇒ match r with
                                                Plus key x y  ⇒ Success (N (x + y))
                                                Minus key x y ⇒ Success (N (x - y))
                                                Not key x     ⇒ Success (B (not x)))
                                                -- Fail case is ruled out
msg : Error → String
msg err = match err with InvalidKeyError ⇒ "invalid key"
```

Fig. 4. BAS Version 4: fully specified datatypes

Observe that `Request` and `Error` are still declared **open**, so they can still be optimistically inhabited with unclassified data, and pattern matches on values of these types can handle arbitrary constructors. Here, for instance, the programmer has added experimental support for a `Not` operation.

Version 4. Finally, BAS Version 4 is almost fully statically typed (Figure 4). Every datatype is **closed**, and all unknown types ? have been replaced with static types. For instance, because the type of `x` is statically declared to be `Int`, the expression `Plus {key=1, x=False, y=7}` is now ill-typed. Similarly, `handleRequest (Times {key=1, x=3, y=7})` now yields a static type error, because `Times` is not a valid constructor of `Request`. By closing all the datatypes, no catch-all cases are needed anymore, which eliminates some runtime errors in pattern matches: the `Error` datatype shrinks accordingly.

But every statically typed language must eventually confront the outside world: `fromJSON` still introduces a term of type ?D. The "parse, don't validate" approach popular in the statically-typed functional programming community suggests writing a wrapper around `fromJSON` to ensure that we only process appropriate `Requests`; we omit that development here to not belabor the point.

Summary. We have illustrated a simple language design for gradually structured data, which combines well-known open/closed datatype declarations with standard gradual types, further enriched with two new gradual types ($?_{\mathrm{D}}$ as the ground type of constructed data, and $?_{\mathrm{O}}$ as the unknown open datatype). The resulting language accommodates a wide range of evolution and type strengthening scenarios, all within the same frame of reference, which seems appealing in practice. The following two sections develop the formalisms underlying GSD: Section 3 presents the static language we consider as a starting point, and Section 4 exposes the gradual language derived from it with AGT [Garcia et al. 2016].

3 ALGEBRAIC DATATYPES, STATICALLY

We begin by describing a statically-typed language λ_{D}: a call-by-value simply-typed lambda calculus with algebraic datatypes. In the literature, models of algebraic datatypes come primarily in two flavors: a simplified algebraic model, with pairs ((e_1, e_2), fst, snd) and disjoint sums (inl, inr, match) and unit (the 0-ary or nullary product, $()$), and a general, flexibly structural model in terms of row types [Wand 1987]. We do not adopt either of these common models, opting instead for an explicit model of named datatypes with associated constructors and a general notion of pattern matching. We are interested in building a model that can simulate some of the dynamics and pragmatics of contemporary languages.

To this end, two key features of λ_{D} are worth highlighting upfront. First, λ_{D} supports *open* datatypes, which can be extended with new constructors. Recall that some statically-typed languages support similar features: Scala has both open and closed variants; OCaml does, too, though most open datatypes (polymorphic variants) are syntactically separated from standard closed datatypes.[2] Second, λ_{D}'s static semantics is parameterized over *matching strategies*, to account for variation in real languages: Haskell does not even *warn* on incomplete matches by default; OCaml lets incomplete matches off with a warning, likewise for Scala with *sealed case classes*; Elm rejects incomplete matches.

3.1 Syntax

The syntax of λ_{D} extends the simply-typed lambda calculus with algebraic datatypes in the style of statically-typed functional languages like Haskell and OCaml (Figure 5).[3] Types T are conventional, including base types B (such as Int or String), function types $T_1 \rightarrow T_2$, and named algebraic datatypes D. Each *datatype name* D is unique. Datatype contexts Δ map datatype names to *constructor sets* C, which may be empty (e.g., the uninhabited "void" type) and their *openness*. Recall that closed datatypes have a fixed set of variants given at definition time, while open ones can be extended with additional variants. Constructor contexts Ξ

[2] OCaml's treats the open datatype of exceptions, exn, specially. It is a an extensible nonrecursive type that uses the conventional constructor syntax. Pattern matches on exn without a wildcard provoke a warning. It would be bad practice to co-opt exn.

[3] Some notational conventions are defined for conciseness. We write $\overline{l} = \overline{e}$ to mean $l_1 = e_1, \ldots, l_n = e_n$ and $\overline{p} \mapsto \overline{e}$ to mean $p_1 \mapsto e_1; \ldots; p_n \mapsto e_n$. When necessary we add a superscript next to the overline to make explicit the number of elements being repeated, eg. $\overline{l}^i = \overline{e}^i$ means $l_1 = e_1, \ldots, l_i = e_i$. Likewise, we write $(\overline{x} : \overline{T})$, or more explicitly $(\overline{x}^n : \overline{T}^n)$, to mean $(x_1 : T_1) \ldots (x_n : T_n)$. We write $\{\overline{p}\}$, or $\{\overline{p}^n\}$, to mean the set $\{p_1, \ \ldots, p_n\}$. In the typing judgments we also differentiate between definitional ($\doteq$) and propositional equality ($=$).

Datatypes

Datatype names	D	$\in$	DTName
Openness	O	$\in$	Open $\mid$ Closed
Constructor names	c	$\in$	CtorName
Constructor sets	C	$\in$	Ctors $\subseteq$ CtorName
Labels	l	$\in$	Label

Contexts

Datatype ctxs	Δ	::=	$\cdot \mid \Delta, D : (C, O)$
Constructor ctxs	Ξ	::=	$\cdot \mid$
			$\Xi, c : (l : T) \times \cdots \times (l : T)$
Type ctxs	Γ	::=	$\cdot \mid \Gamma, x : T$

Types

Base types	B	::=	Int $\mid \ldots$
Types	T	::=	$B \mid T \to T \mid D$

Terms

Expressions	e	::=	$x \mid k \mid e\ e \mid \lambda x{:}T.\ e$
		$\mid$	$e :: T \mid c\ \left\{\overline{l} = \overline{e}\right\}$
		$\mid$	$e.l \mid$ match e with $\{\overline{p} \mapsto \overline{e}\}$
Identifiers	x	$\in$	Var
Constants	k	::=	$0 \mid 1 \mid \ldots \mid + \mid \ldots$
Patterns	p	::=	$c\ \overline{x}$
Values	v	::=	$k \mid \lambda x{:}T.\ e \mid c\ \left\{\overline{l} = \overline{v}\right\}$
Errors	**err**	::=	$\mathbf{error_M} \mid \mathbf{error_A}$

Fig. 5. λ_{D} syntax

map *constructor names* c to ordered, labeled products of types, i.e., records. Record labels l should be used at most once for each constructor, but can be reused across them.

The expressions of the language include the usual variables, constants k, lambda abstraction, applications, and static type ascriptions (which play a key role in the gradualization with AGT). We add three syntactic forms to support datatypes: constructor applications, field accesses, and pattern matching.

The *constructor application* $c\ \{l_1 = e_1, \ldots, l_n = e_n\}$ applies the constructor c with field l_i set to (the value of) e_i. The *field access* $e.l$ extracts field l from the data value resulting from evaluating e. Finally, a *pattern match* match e with $\{p_1 \mapsto e_1; \ldots; p_n \mapsto e_n\}$ compares the value of the discriminee e against each pattern p_i in turn; when a pattern matches, the parts of the discriminee are bound by p_i, and e_i is evaluated with these bindings. As a modeling compromise, we do not allow nested patterns in the formalism.

We include field access explicitly because it is (a) a useful feature in real languages, but (b) challenging to encode faithfully using only pattern matching in the presence of open datatypes.

3.2 Static Semantics

The static language λ_{D} enjoys a mostly conventional static semantics, with ordinary rules for type well-formedness, type equality, and term typing (Figure 6). Anticipating the use of AGT to derive the gradual language, we define the static semantics a little more abstractly than usual: side conditions in rules are explicit predicates and partial functions (Figure 7), so that we can use Galois connections to derive the gradual versions (Section 4.1). The judgments concerning types themselves are conventional, requiring that datatypes be well formed in the datatype context Δ, which in turn requires that Δ be well formed. Context well-formedness is formally complicated—we build up three contexts well-formedness judgments on top of each other. Despite the formal complexity, the context rules are conceptually simple. First, the datatype context Δ is well formed when each datatype is assigned a disjoint set of constructors (Δ-Ext). Next, given

Typing rules (standard STLC rules omitted) $\boxed{\Delta; \Xi; \Gamma \vdash e : T}$

$$\frac{\Delta; \Xi; \Gamma \vdash e : T \qquad T = T'}{\Delta; \Xi; \Gamma \vdash e :: T' : T'}\ \textsc{T-Ascribe} \qquad \frac{\begin{array}{c} T \doteq \mathsf{cty}_\Delta(c) \qquad \mathsf{isdata}_\Delta(T) \qquad \Delta; \Xi; \Gamma \vdash \overline{e} : \overline{T} \\ \overline{T} = \mathsf{lty}_\Xi(\overline{l}, c) \qquad \mathsf{satisfylabels}_{n\Xi}(c, l_1 \times \cdots \times l_n) \end{array}}{\Delta; \Xi; \Gamma \vdash c\ \left\{\overline{l}^n = \overline{e}^n\right\} : T}\ \textsc{T-Ctor}$$

$$\frac{\Delta; \Xi; \Gamma \vdash e : T \qquad \mathsf{isdata}_\Delta(T)}{\Delta; \Xi; \Gamma \vdash e.l : \mathsf{fty}_{\Delta,\Xi}(l, T)}\ \textsc{T-Access} \qquad \frac{\begin{array}{c} \Delta; \Xi; \Gamma \vdash e : T \qquad \mathsf{isdata}_\Delta(T) \qquad \mathsf{valid}_{\Delta,\Xi}(\{\overline{p}\}, T) \\ (\forall i.1 \leq i \leq n) \qquad (x_{i1} : T_{i1}) \times \cdots \times (x_{im_i} : T_{im_i}) \doteq \mathsf{parg}_{m_i\Xi}(p_i) \\ \Delta; \Xi; \Gamma, x_{i1} : T_{i1}, \ \ldots, \ x_{im_i} : T_{im_i} \vdash e_i : T_i \end{array}}{\Delta; \Xi; \Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ \{\overline{p}^n \mapsto \overline{e}^n\} : \mathsf{equate}_{n\Delta}(T_1, \ \ldots, \ T_n)}\ \textsc{T-Match}$$

Reductions $\boxed{e \longrightarrow e \text{ or } \mathbf{err}}$

$$\begin{array}{llcl}
\text{(R-Beta)} & (\lambda x{:}T.\ e)\ v & \longrightarrow & e\,[{}^{v}/_{x}] \\
\text{(R-Delta)} & k\ v & \longrightarrow & \delta(k, v) \\
\text{(R-AscErase)} & v :: T & \longrightarrow & v \\
\text{(R-Match)} & \mathsf{match}\ c\ \left\{\overline{l} = \overline{v}\right\}\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\} & \longrightarrow & \begin{cases} e_k\,[{}^{\overline{v}}/_{\overline{x_k}}] & k \text{ smallest } \mathbb{N} \text{ s.t. } c = c_k \text{ where } c_k\ \overline{x_k} \doteq p_k \\ \mathbf{error_M} & \text{there is no } k \text{ s.t. } c = c_k \end{cases} \\
\text{(R-Access)} & \left(c\ \left\{\overline{l} = \overline{v}\right\}\right).l & \longrightarrow & \begin{cases} v_k & l_k = l \\ \mathbf{error_A} & \text{otherwise} \end{cases}
\end{array}$$

Fig. 6. λ_D: typing and dynamic semantics

a datatype context Δ, the constructor context Ξ is well formed (Ξ-Ext) when each constructor (a) belongs to some datatype D (cty_Δ), (b) has arguments with disjoint labels, (c) each label is mapped to a well-formed, closed type, and (d) every other constructor in the same datatype that uses the same label uses the same type at that label. Finally, given a datatype context Δ and constructor context Ξ, the type context Γ is well formed if the types inside of it are well formed (Γ-Ext).

The term typing rules for λ_D are standard (Figure 6). We omit the usual lambda calculus rules for brevity and focus on the rules specific to datatypes—T-Ctor, T-Access and T-Match—as well as the ascription rule T-Ascribe. Constructor application is well-typed (T-Ctor) when (a) its type is a datatype, (b) the labels l_i match what we know about the constructor, and (c) the subterms at each label are of the correct type for that label. A field access $e.l$ is well-typed (T-Access) when the type of e is a datatype with at least one constructor for which l is valid. Pattern matches are well-typed (T-Match) when (a) the type of the term being matched is a datatype, (b) the branches have the same type in a context extended with the type of the pattern bindings, and (c) the patterns are valid with respect to the type of the term being matched. λ_D's static semantics accommodates different interpretations of when pattern matches are valid—that is, when a list of patterns is considered sufficiently exhaustive (see below).

The typing rules use a number of partial type functions (Figure 7): $\mathsf{equate}_{n\Delta}$ computes the *meet* of a collection of types (used in T-Match); cty_Δ computes the type of a constructor (used in Ξ-Ext and T-Ctor); $\mathsf{parg}_{n\Xi}$ computes the bindings, with their expected types, of a pattern (used in T-Match); $\mathsf{fty}_{\Delta,\Xi}$ computes

$$
\begin{array}{lll}
\mathsf{ctors}_\Delta : \mathsf{Type} \rightharpoonup \mathcal{P}(\mathsf{Ctors}) & & \mathsf{cty}_\Delta : \mathsf{Ctors} \rightharpoonup \mathsf{Type} \\
\mathsf{ctors}_\Delta(D) \;=\; \pi_1(\Delta(D)) & & \mathsf{cty}_\Delta(c) \;=\; \bigsqcap \{D \in \mathsf{dom}(\Delta) \mid c \in \mathsf{ctors}_\Delta(D)\} \\
\mathsf{ctors}_\Delta(\_) \;=\; \bot & & \\
 & & \mathsf{lty}_\Xi : \mathsf{Label} \times \mathsf{Ctors} \rightharpoonup \mathsf{Type} \\
\mathsf{pctor} : \mathsf{Pattern} \to \mathsf{Ctors} & & \mathsf{lty}_\Xi(l, c) \;=\; T_i \quad (l_1 : T_1) \times \cdots \times (l_n : T_n) = \Xi(c) \wedge l = l_i \\
\mathsf{pctor}(c\ \overline{x}) \;=\; c & & \mathsf{lty}_\Xi(l, c) \;=\; \bot \quad \text{otherwise} \\
 & & \\
\mathsf{equate}_{n\Delta} : \mathsf{Type}^n \rightharpoonup \mathsf{Type} & & \mathsf{fty}_{\Delta,\Xi} : \mathsf{Label} \times \mathsf{Type} \rightharpoonup \mathsf{Type} \\
\mathsf{equate}_{n\Delta}(T, \ldots, T) \;=\; T & & \mathsf{fty}_{\Delta,\Xi}(l, D) \;=\; \bigsqcap \left\{\mathsf{lty}_\Xi(l, c) \mid c \in \mathsf{ctors}_\Delta(D)\right\} \\
\mathsf{equate}_{n\Delta}(\_, \ldots, \_) \;=\; \bot & & \mathsf{fty}_{\Delta,\Xi}(l, T) \;=\; \bot
\end{array}
$$

$$
\mathsf{parg}_{n\Xi} : \mathsf{Pattern} \rightharpoonup (\mathsf{Var} : \mathsf{Type})^n
$$

$$
\mathsf{parg}_{n\Xi}(c\ (\overline{x}^n)) = \begin{cases} (x_1 : T_1) \times \cdots \times (x_n : T_n) & (l_1 : T_1) \times \cdots \times (l_m : T_m) \doteq \Xi(c) \wedge n = m \\ \bot & c \notin \mathsf{dom}(\Xi) \vee n \neq m \end{cases}
$$

$$
\mathsf{satisfylabels}_{n\Xi}(c, l_1 \times \cdots \times l_n) \Leftrightarrow \mathsf{permutation}(l_1 \ldots l_n, l'_1 \ldots l'_m) \quad (l'_1 : T_1) \times \cdots \times (l'_m : T_m) \doteq \Xi(c)
$$

$$
\mathsf{valid}_{\Delta,\Xi}(P, D) \Leftrightarrow \bigcup_{p \in P} \mathsf{pctor}(p) \star \mathsf{ctors}_\Delta(D) \quad \text{where } \star \text{ is: } \subseteq \text{ if Sound, } = \text{ if Exact, or } \supseteq \text{ if Complete.}
$$

$$
\mathsf{isdata}_\Delta(T) \Leftrightarrow \exists D \in \mathsf{dom}(\Delta),\ D = T \qquad \mathsf{open}_\Delta(D) \Leftrightarrow \pi_2(\Delta(D)) = \mathsf{Open}
$$

Fig. 7. λ_{D}: type functions and predicates for datatype-related rules

the type a label has for a specific type (used in Ξ-Ext and T-Access); lty_Ξ computes the type of a label for a specific constructor (used in T-Ctor and $\mathsf{fty}_{\Delta,\Xi}$); ctors_Δ computes the constructor set of a datatype (used in Δ-Ext, $\mathsf{fty}_{\Delta,\Xi}$, cty_Δ and $\mathsf{valid}_{\Delta,\Xi}$); We also use one total function: pctor computes the constructor name of a pattern (used in T-Match via $\mathsf{valid}_{\Delta,\Xi}$; see below); We use some type predicates beyond the type equality in T-Ascribe: $\mathsf{satisfylabels}_{n\Xi}$ checks that a constructor has a specific set of labels (used in T-Ctor); isdata_Δ checks if a type is a datatype (used in T-Ctor, T-Access and T-Match); and, finally, $\mathsf{valid}_{\Delta,\Xi}$ determines whether or not a set of patterns is sufficient (used in T-Match).

Matching strategies. The language λ_{D} is parametric with respect to the meaning of valid pattern matches. We introduce three possible *matching strategies* that characterize valid matches ($\mathsf{valid}_{\Delta,\Xi}$, Figure 7): sound, complete, and exact. *Sound* matching corresponds to Haskell's policy: every pattern in the match expression must correspond to at least one constructor in the datatype, but partial matches are allowed. *Complete* matching does not allow for partial matches, but case branches can have patterns that do not match any constructor in the type. We are not aware of any statically-typed language that lets one write extra cases using impossible constructor names. Even so, allowing dead branches is harmless in the static system, and as we will see, the choice proves advantageous as we gradualize the type system. *Exact* matching demands that every single constructor is accounted for, with none missing and none extra. The exact regime is used for instance in Coq and Elm. Therefore, exact matching is both *sound* and *complete*.

For example, using the concrete syntax of our implementation, the pattern matches below are sound exact and complete with respect to A respectively:

```
data A = Foo | Bar
data B = Baz
```

```
-- Sound
match x with
  Foo ⇒ ...
```

```
-- Exact
match x with
  Foo ⇒ ...
  Bar ⇒ ...
```

```
-- Complete
match x with
  Foo ⇒ ...
  Bar ⇒ ...
  Baz ⇒ ...
```

As we will see, under any matching strategy, AGT allows us to derive meaningful gradual variants of λ_{D}.

3.3 Dynamic Semantics

The reduction rules of λ_{D} are mostly standard (Figure 6). We use evaluation frames with call-by-value semantics with errors; subterms are reduced from left to right and evaluation may fail. Values are the usual suspects: constants, functions, and constructors applied to values (Figure 5).

There are five reduction rules: R-Beta reduces a function application by substitution, R-Delta reduces an operator application by means of the δ function (which we assume to be adequately typed), R-AscErase drops ascriptions on values, R-Match performs pattern matching on datatype values, and R-Access performs partial field access on a constructor.

Note that field access is partial—i.e., it can fail at runtime, producing $\mathbf{error_A}$. Field access $e.l$ fails when e reduces to a constructor without l. To illustrate this, suppose we define a datatype of integer lists ($\mathsf{List} : \{\mathsf{Cons}, \mathsf{Nil}\} \in \Delta$ and $\{\mathsf{Cons} : (hd : \mathsf{Int}) \times (tl : \mathsf{List}), \mathsf{Nil} : \langle\rangle\} \subseteq \Xi$). Then $\mathsf{Nil}.hd$ would be well typed but fail at runtime, because Nil does not have hd as a field.

Depending on which matching strategy is used to typecheck the program ($\mathsf{valid}_{\Delta,\Xi}$, Section 3.2), pattern matching may fail at runtime. Specifically, complete and exact pattern matching are guaranteed not to fail at runtime, but sound pattern matching might, producing $\mathbf{error_M}$.

3.4 Metatheory

The exact type safety property that λ_{D} enjoys depends on the choice of matching strategy. As we have just seen, in the exact and complete interpretations of $\mathsf{valid}_{\Delta,\Xi}$, programs can only fail with errors caused by a field access error ($\mathbf{error_A}$). But, if $\mathsf{valid}_{\Delta,\Xi}$ is only meant to be sound, programs can also fail by a pattern match error ($\mathbf{error_M}$). To encode this dependency, we define the function errors, which maps a matching strategy $m \in \{\mathsf{Sound}, \mathsf{Exact}, \mathsf{Complete}\}$ to the set of errors that might be produced: $\mathsf{errors}(\mathsf{Sound}) = \{\mathbf{error_M}, \mathbf{error_A}\}$ and $\mathsf{errors}(m) = \{\mathbf{error_A}\}$ otherwise.

Theorem 3.1 (Type safety of λ_{D}). *When using matching strategy m, if $\Delta; \Xi; \cdot \vdash e : T$ then either $e \Downarrow v$ with $\Delta; \Xi; \cdot \vdash v : T$, or $e \Downarrow \mathbf{err}$ where $\mathbf{err} \in \mathsf{errors}(m)$.*

$$
\begin{array}{rcl}
\multicolumn{3}{c}{\gamma_\Delta : \mathsf{GType} \to \mathcal{P}^*(\mathsf{Type})} \\
\gamma_\Delta(B) & = & \{B\} \\
\gamma_\Delta(G_1 \to G_2) & = & \{T_1 \to T_2 \mid T_1 \in \gamma_\Delta(G_1), \\
& & \qquad\qquad T_2 \in \gamma_\Delta(G_2)\} \\
\gamma_\Delta(D) & = & \{D\} \\
\gamma_\Delta(?_\mathrm{O}) & = & \left\{D \in \mathsf{dom}(\Delta) \mid \mathsf{open}_\Delta(D)\right\} \\
\gamma_\Delta(?_\mathrm{D}) & = & \{D \in \mathsf{dom}(\Delta)\} \\
\gamma_\Delta(?) & = & \mathsf{Type}
\end{array}
$$

$$
\begin{array}{rcl}
\multicolumn{3}{c}{\alpha_\Delta : \mathcal{P}^*(\mathsf{Type}) \to \mathsf{GType}} \\
\alpha_\Delta(\{B\}) & = & B \\
\alpha_\Delta\left(\left\{\overline{T_{i1} \to T_{i2}}\right\}\right) & = & \alpha_\Delta\left(\left\{\overline{T_{i1}}\right\}\right) \to \alpha_\Delta\left(\left\{\overline{T_{i2}}\right\}\right) \\
\alpha_\Delta(\{D\}) & = & D \\
\alpha_\Delta\left(\left\{\overline{D_i}\right\}\right) & = & \begin{cases} ?_\mathrm{O} & \forall i.\ \mathsf{open}_\Delta(D_i) \\ ?_\mathrm{D} & \text{otherwise} \end{cases} \\
\alpha_\Delta\left(\left\{\overline{T_i}\right\}\right) & = & ?
\end{array}
$$

Type Precision (reflexive rules omitted) $\boxed{\Delta \vdash G \sqsubseteq G}$

$$
\frac{\Delta \vdash G_{11} \sqsubseteq G_{21} \qquad \Delta \vdash G_{21} \sqsubseteq G_{22}}{\Delta \vdash G_{11} \to G_{12} \sqsubseteq G_{21} \to G_{22}}\ \text{P-Arrow}
\qquad
\frac{}{\Delta \vdash G \sqsubseteq ?}\ \text{P-?}
$$

$$
\frac{\Delta \vdash D}{\Delta \vdash D \sqsubseteq ?_\mathrm{D}}\ \text{P-}?_\mathrm{D}\text{R}
\qquad
\frac{}{\Delta \vdash ?_\mathrm{O} \sqsubseteq ?_\mathrm{D}}\ \text{P-}?_\mathrm{O}\text{L}
\qquad
\frac{\Delta \vdash D \qquad \mathsf{open}_\Delta(D)}{\Delta \vdash D \sqsubseteq ?_\mathrm{O}}\ \text{P-}?_\mathrm{O}\text{R}
$$

Type Consistency (reflexive and right side of symmetric rules omitted) $\boxed{\Delta \vdash G \sim G}$

$$
\frac{\Delta \vdash G_1 \sim G_1' \qquad \Delta \vdash G_2 \sim G_2'}{\Delta \vdash G_1 \to G_2 \sim G_1' \to G_2'}\ \text{C-Arrow}
\qquad
\frac{}{\Delta \vdash ? \sim G}\ \text{C-?L}
$$

$$
\frac{\Delta \vdash D}{\Delta \vdash D \sim ?_\mathrm{D}}\ \text{C-DataL}
\qquad
\frac{}{\Delta \vdash ?_\mathrm{D} \sim ?_\mathrm{O}}\ \text{C-}?_\mathrm{D}\text{L}
\qquad
\frac{\Delta \vdash D \qquad \mathsf{open}_\Delta(D)}{\Delta \vdash ?_\mathrm{O} \sim D}\ \text{C-}?_\mathrm{O}\text{L}
$$

Fig. 8. Concretization and abstraction functions, type precision and type consistency

4 GRADUALLY STRUCTURED DATA

We now gradualize λ_D following the AGT methodology [Garcia et al. 2016], yielding the gradual language $\lambda_{\mathrm{D}?}$. This section focuses on the statics of $\lambda_{\mathrm{D}?}$. We first introduce gradual types and the induced notion of (im)precision, and then lift the statics semantics of λ_D to account for imprecise types. We then develop the runtime semantics of $\lambda_{\mathrm{D}?}$ via an intermediate language, and conclude with the metatheory of $\lambda_{\mathrm{D}?}$.

4.1 Semantics of gradual types

The first step in building $\lambda_{\mathrm{D}?}$ is to define a denotation of gradual types as static types; this denotation is called a *concretization*. Equipped with concretization, one can then lift static type predicates (such as equality) to operate on gradual types (yielding consistency). Then, using the dual operation of *abstraction*, static type functions (like dom, $\mathsf{equate}_{n\Delta}$, and $\mathsf{valid}_{\Delta,\Xi}$) can be lifted, yielding the gradual type system.

The concretization function γ_Δ gives meaning to a gradual type by producing the non-empty set of static types that it denotes (Figure 8). Gradual types are noted by the metavariable G:

$$G ::= B \mid G \to G \mid D \mid ? \mid ?_\mathrm{D} \mid ?_\mathrm{O}$$

As usual, the unknown type denotes any static type. We deal with gradually structured data, and so new forms of gradual types are required. First, we introduce the unknown datatype $?_\mathrm{D}$, which stands for any datatype D. Values of type $?_{\mathsf{D}}$ can be eliminated through pattern matching and field access. We say that $?_{\mathsf{D}}$ is the *ground type* for datatypes, both open and closed, just like $? \rightarrow ?$ is the ground type of functions, which can be eliminated through application. Additionally, to support gradually structured data, it is helpful to account for "free-floating" constructors, which do not (yet) belong to any statically defined datatype: the unknown open datatype $?_{\mathsf{O}}$, represents any *open* datatype.

Gradual types are drawn from static types and type constructors together with the three unknown types. As expected, a static type like Int or $\mathsf{Int} \rightarrow \mathsf{Bool}$ is a fully precise gradual type, which only denotes itself. In our setting, concretization is indexed by the datatype context Δ, since it is required to give meaning to both the unknown datatype $?_{\mathsf{D}}$ and the unknown open datatype $?_{\mathsf{O}}$. Technically, for concretization to be well-defined, Δ must include at least one open datatype, otherwise $\gamma_\Delta(?_{\mathsf{D}})$ and $\gamma_\Delta(?_{\mathsf{O}})$ would denote the empty set of static types. The syntax of Δ is modified to reflect this requirement:

$$\Delta ::= D : (C, \mathsf{Open}) \ \mid \ \Delta, D : (C, O) \ \mid \ \Delta, ?_{\mathsf{O}} : (C, \mathsf{Open})$$

Furthermore, unclassified data is stored in Δ as a constructor of the open type $?_{\mathsf{O}}$.

Because several type-level functions and predicates are indexed by contexts, we need to define concretization for both datatype and constructor contexts. The concretization functions (in supplementary material) take some context with gradual types and compute the set of static contexts that the gradual context represents. The concretization of a gradual datatype context produces a set of static contexts in which unclassified data (i.e., constructors appearing in $?_{\mathsf{O}}$) are added to the constructor set of each open datatype. The concretization of a gradual constructor context is the pointwise concretization of the types in each constructor's definition. Hereafter, we mark static contexts with a subscript S, such as Δ_S, to avoid ambiguities.

Having defined the concretization of gradual types, one can derive the notion of *precision* between gradual types: a gradual type G_1 is less precise than another gradual type G_2 if its concretization is a superset of that of G_2. Note that because concretization relies on Δ, the precision judgment is contextual.

Definition 4.1 (Type Precision). $\Delta \vdash G \sqsubseteq G'$ if and only if $\gamma_\Delta(G) \subseteq \gamma_\Delta(G')$.
An equivalent inductive definition is given in Figure 8.

Note that for any open datatype D, we have $\Delta \vdash D \sqsubseteq ?_{\mathsf{O}} \sqsubseteq ?_{\mathsf{D}} \sqsubseteq ?$. On the other hand, $?_{\mathsf{O}}$ and $? \rightarrow ?$ are unrelated by precision.

Armed with concretization, the AGT framework gives us a direct way to lift type predicates from the static language: a predicate holds on gradual types if there *exist* types in the respective concretizations that satisfy the static predicate. For instance, the type equality judgment is lifted into type consistency. Two gradual types are said to be consistent if they have at least one static type in common in their denotations.

Definition 4.2 (Type consistency). $\Delta \vdash G \sim G' \Leftrightarrow \exists T \in \gamma_\Delta(G), \exists T' \in \gamma_\Delta(G'), T = T'$.
An equivalent inductive definition is given in Figure 8.

In addition to type equality, the statics of λ_{D} use two predicates specific to datatypes: isdata_Δ holds if a given type is a datatype, and $\mathsf{valid}_{\Delta,\Xi}$ holds if a match expression is valid for a given type (recall that we consider three matching strategies, Figure 7). The consistent lifting of isdata_Δ is simply consistency with respect to $?_{\mathrm{D}}$: $\widetilde{\mathsf{isdata}_\Delta}(G) \Leftrightarrow \Delta \vdash G \sim ?_{\mathrm{D}}$.

The consistent lifting of $\mathsf{valid}_{\Delta,\Xi}$ is $\widetilde{\mathsf{valid}_{\Delta,\Xi}}(P, G) \Leftrightarrow \exists \Delta_S \in \gamma(\Delta),\ \Xi_S \in \gamma_\Delta(\Xi),\ T \in \gamma_\Delta(G),\ \mathsf{valid}_{\Delta,\Xi}(P, T)$. A match is valid if there exists some type in the concretization of the type of the discriminee for which it is valid. In practice this means that when matching on a term whose type is gradual, patterns must be valid for any datatype, in the case of ? or $?_{\mathrm{D}}$, or any *open* datatype, for $?_{\mathrm{O}}$. Moreover, if one wants to make a match valid with respect to some open datatype (or $?_{\mathrm{O}}$), patterns must take into account unclassified data also. On the other hand, if the type of the term being matched is static, every matching strategy behaves in the same way as in the static language (Exact and Complete matches cannot fail). Importantly, in contrast to the static language, no strategy can avoid matching errors. For example, given two closed datatypes A and B, each with a single constructor named A and B respectively, the expression $\mathsf{match}\ (A :: ?)\ \mathsf{with}\ \{B \mapsto 0\}$ both typechecks and fails at runtime for any matching strategy, because it will be $\mathsf{valid}_{\Delta,\Xi}$ for B.

Finally, to be able to lift type functions to operate on gradual types, we need to define abstraction as the counterpart of concretization. Given a set of static types, abstraction yields the most precise gradual type that denotes (at least) this set; its definition is straightforward (Figure 8). We can then establish that abstraction is both sound and optimal, yielding a Galois connection, an important property for AGT-derived languages. Equipped with abstraction, type functions such as dom and $\mathsf{equate}_{n\Delta}$ are lifted into their consistent counterparts by first concretizing their inputs and abstracting the collection of their possible outputs. For example the consistent lifting of the function cty_Δ is: $\widetilde{\mathsf{cty}_\Delta}(c) = \bigsqcap \{G \in \mathsf{dom}(\Delta) \ \mid\ c \in \mathsf{ctors}_\Delta(G)\}$ where the meet between gradual types is defined as $G_1 \sqcap G_2 = \alpha_\Delta(\gamma_\Delta(G_1) \cap \gamma_\Delta(G_2))$ [Garcia et al. 2016]. An algorithmic inductive definition of $\sqcap$ can be found in Appendix.

4.2 Static semantics of $\lambda_{\mathrm{D}?}$

Armed with the definitions of type predicates and functions systematically derived from the meaning of gradual types, the static semantics of the gradual language $\lambda_{\mathrm{D}?}$ follow directly. The typing rules (Figure 9) mirror the rules of the static language, except that all predicates and functions are replaced by their consistent counterparts. For example, $\widetilde{\mathsf{lty}_\Xi}$ computes the gradual type of label for a given constructor, and $\widetilde{\mathsf{equate}_{n\Delta}}$ computes the meet of its inputs. The only interesting rules are G-Ascribe, G-Ctor, G-Access and G-Match. An ascription $e :: G$ is well-typed (G-Ascribe) when the type of e is consistent with G. Constructor application is well-typed (G-Ctor) when (a) its type is *consistent* with a datatype, (b) the labels l_i match what we know about the constructor, and (c) the subterms at each label have a type that is consistent with the correct type for that label. A field access $e.l$ is well-typed (G-Access) when the type of e is consistent with a datatype with at least one constructor for which l is valid.

Typing rules $\boxed{\Delta; \Xi; \Gamma \vdash e : G}$

$$\frac{\Delta; \Xi \vdash \Gamma \qquad G \doteq \Gamma(x)}{\Delta; \Xi; \Gamma \vdash x : G} \text{ G-VAR} \qquad \frac{}{\Delta; \Xi; \Gamma \vdash k : \mathsf{ty}(k)} \text{ G-CONST}$$

$$\frac{\Delta; \Xi; \Gamma \vdash e_1 : G \qquad \Delta; \Xi; \Gamma \vdash e_2 : G' \qquad \Delta \vdash G' \sim \widetilde{\mathsf{dom}}_\Delta(G)}{\Delta; \Xi; \Gamma \vdash e_1\ e_2 : \widetilde{\mathsf{cod}}_\Delta(G)} \text{ G-APP}$$

$$\frac{\Delta; \Xi; \Gamma, x : G_x \vdash e : G}{\Delta; \Xi; \Gamma \vdash \lambda x{:}G_x.\ e : G_x \to G} \text{ G-LAM} \qquad \frac{\Delta; \Xi; \Gamma \vdash e : G \qquad \Delta \vdash G \sim G'}{\Delta; \Xi; \Gamma \vdash e :: G' : G'} \text{ G-ASCRIBE}$$

$$\frac{\begin{array}{c} G \doteq \widetilde{\mathsf{cty}}_\Delta(c) \qquad \Delta \vdash G \sim ?_D \qquad \Delta; \Xi; \Gamma \vdash \overline{e} : \overline{G} \\ \Delta \vdash \overline{G} \sim \widetilde{\mathsf{lty}}_\Xi(\overline{l}, c) \qquad \mathsf{satisfylabels}_{n\Xi}(c, l_1 \times \cdots \times l_n) \end{array}}{\Delta; \Xi; \Gamma \vdash c\ \left\{\overline{l}^n = \overline{e}^n\right\} : G} \text{ G-CTOR} \qquad \frac{\Delta; \Xi; \Gamma \vdash e : G \qquad \Delta \vdash G \sim ?_D}{\Delta; \Xi; \Gamma \vdash e.l : \widetilde{\mathsf{fty}}_{\Delta,\Xi}(l, G)} \text{ G-ACCESS}$$

$$\frac{\begin{array}{c} \Delta; \Xi; \Gamma \vdash e : G \qquad \Delta \vdash G \sim ?_D \qquad \widetilde{\mathsf{valid}}_{\Delta,\Xi}(\{\overline{p}\}, G) \\ (\forall i. 1 \le i \le n) \qquad (x_{i1} : G_{i1}) \times \cdots \times (x_{im_i} : G_{im_i}) \doteq \widetilde{\mathsf{parg}}_{m_i\Xi}(p_i) \\ \Delta; \Xi; \Gamma, x_{i1} : G_{i1}, \ \ldots, \ x_{im_i} : G_{im_i} \vdash e_i : G_i \end{array}}{\Delta; \Xi; \Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ \{\overline{p}^n \mapsto \overline{e}^n\} : \widetilde{\mathsf{equate}}_{n\Delta}(G_1, \ldots, G_n)} \text{ G-MATCH}$$

Fig. 9. $\lambda_{D?}$: Static semantics

Pattern matches are well-typed (G-Match) when (a) the type of the term being matched is consistent with a datatype, (b) the types of the branches in a context extended with the type of the pattern bindings are consistent with each other, and (c) the patterns are valid with respect to the type of the term being matched.

Typing appeals to an extended notion of type well-formedness, which is straightforward. The well-formedness of contexts follows from the static definition, using the lifted version of functions and predicates. Apart from these changes, the well-formedness of Δ is modified to take into account its new syntax. Δ is said to be well-formed if it has at least one open datatype and every type in the domain of Δ has a disjoint set of constructors. Apart from datatypes, $?_O$ may also appear in the domain of Δ, and its constructors are what we called unclassified data.

4.3 Dynamic semantics of $\lambda_{D?}$

In order to define the dynamic semantics of $\lambda_{D?}$, we follow the approach of Toro et al. [2019] and introduce an auxiliary language $\lambda^{\varepsilon}_{D?}$, which is just a variant of $\lambda_{D?}$ in which every ascription carries the *evidence* that supports its validity. Evaluation of $\lambda_{D?}$ consists of an elaboration to $\lambda^{\varepsilon}_{D?}$ (akin to a cast insertion translation [Siek and Taha 2006], but following the AGT framework), and then the actual reduction of the elaborated $\lambda^{\varepsilon}_{D?}$ term. We first explain the syntax, static and dynamic semantics of $\lambda^{\varepsilon}_{D?}$, and then come back to the evaluation of $\lambda_{D?}$ and complete its metatheory.

$$
\begin{array}{lrl}
\textit{Evidence} & \varepsilon & \in\ \mathsf{GType} \\
\textit{Expressions} & e & ::= x \mid k \mid e\ e \mid \lambda x{:}G.\ e \mid \varepsilon\ e :: G \\
 & & \mid\ c\ \left\{\overline{l} = \overline{e}\right\} \mid \mathsf{match}\ e\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\}
\end{array}
\qquad
\begin{array}{lrl}
\textit{Values} & v & ::= \varepsilon\ u :: G \\
\textit{Raw Values} & u & ::= k \mid \lambda x{:}G.\ e \mid c\ \left\{\overline{l} = \overline{v}\right\} \\
\textit{Errors} & \mathbf{err} & ::= \mathbf{error_M} \mid \mathbf{error_A} \mid \mathbf{error_T}
\end{array}
$$

Typing rules (variable and constant rules omitted) $\boxed{\Delta; \Xi; \Gamma \vdash e : G}$

$$
\frac{\Delta; \Xi; \Gamma, x : G_x \vdash e : G}{\Delta; \Xi; \Gamma \vdash \lambda x{:}G_x.\ e : G_x \to G}\ \varepsilon\text{G-Lam}
\qquad
\frac{\Delta; \Xi; \Gamma \vdash e_1 : G \qquad \Delta; \Xi; \Gamma \vdash e_2 : \widetilde{\mathsf{dom}}_\Delta(G)}{\Delta; \Xi; \Gamma \vdash e_1\ e_2 : \widetilde{\mathsf{cod}}_\Delta(G)}\ \varepsilon\text{G-App}
$$

$$
\frac{\Delta; \Xi; \Gamma \vdash e : G \qquad \varepsilon \Vdash \Delta \vdash G \sim G'}{\Delta; \Xi; \Gamma \vdash \varepsilon\ e :: G' : G'}\ \varepsilon\text{G-Ascribe}
\qquad
\frac{\begin{array}{c}\Delta; \Xi; \Gamma \vdash \overline{e} : \widetilde{\mathsf{lty}}_\Xi(\overline{l}, c) \\ \widetilde{\mathsf{satisfylabels}}_{n\Xi}(c, l_1 \times \cdots \times l_n)\end{array}}{\Delta; \Xi; \Gamma \vdash c\ \left\{\overline{l}^n = \overline{e}^n\right\} : \widetilde{\mathsf{cty}}_\Delta(c)}\ \varepsilon\text{G-Ctor}
$$

$$
\frac{\Delta; \Xi; \Gamma \vdash e : G \qquad \Delta \vdash G \sqsubseteq ?_\mathrm{D}}{\Delta; \Xi; \Gamma \vdash e.l : \widetilde{\mathsf{fty}}_{\Delta,\Xi}(l, G)}\ \varepsilon\text{G-Access}
\qquad
\frac{\begin{array}{c}\Delta; \Xi; \Gamma \vdash e : G' \qquad \Delta \vdash G' \sqsubseteq ?_\mathrm{D} \qquad \widetilde{\mathsf{valid}}_{\Delta,\Xi}(\{\overline{p}\}, G') \\ \forall i \in [1, n]\ (x_{i1} : G_{i1}) \times \cdots \times (x_{im_i} : G_{im_i}) \doteq \mathsf{parg}_{m_i\Xi}(p_i) \\ \Delta; \Xi; \Gamma, x_{i1} : G_{i1},\ \ldots,\ x_{im_i} : G_{im_i} \vdash e_i : G\end{array}}{\Delta; \Xi; \Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\} : G}\ \varepsilon\text{G-Match}
$$

Reductions $\boxed{e \longrightarrow e \text{ or } \mathbf{err}}$

$$
\begin{array}{rll}
(\varepsilon\text{R-Beta}) & (\varepsilon_1\ (\lambda x{:}G_{11}.\ e) :: G_1 \to G_2)\ (\varepsilon_2\ u :: G_1) & \longrightarrow \begin{cases} \mathsf{cod}(\varepsilon_1)(e\left[{}^{((\varepsilon_2 \circ \mathsf{dom}(\varepsilon_1))\ u\ ::\ G_{11})}/_{x}\right]) :: G_2 & \\ \mathbf{error_T} & \text{if } \varepsilon_2 \circ \mathsf{dom}(\varepsilon_1) \text{ is not defined} \end{cases} \\
(\varepsilon\text{R-Delta}) & (\varepsilon_1\ k :: G_1 \to G_2)\ (\varepsilon_2\ u :: G_1) & \longrightarrow \mathsf{cod}(\varepsilon_1)\ \delta(k, u) :: G_2 \\
(\varepsilon\text{R-AscErase}) & \varepsilon_2\ (\varepsilon_1\ u :: G_1) :: G_2 & \longrightarrow \begin{cases} (\varepsilon_1 \circ \varepsilon_2)\ u :: G_2 & \\ \mathbf{error_T} & \text{if } \varepsilon_1 \circ \varepsilon_2 \text{ is not defined} \end{cases} \\
(\varepsilon\text{R-Match}) & \mathsf{match}\ \varepsilon\ c\ \left\{\overline{l} = \overline{v}\right\} :: G\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\} & \longrightarrow \begin{cases} e_k\ [\overline{v}/\overline{x_k}] & k \text{ smallest } \mathbb{N} \text{ s.t. } c = c_k \\ & \text{where } c_k\ \overline{x_k} \doteq p_k \\ \mathbf{error_M} & \text{there is no } k \text{ s.t. } c = c_k \end{cases} \\
(\varepsilon\text{R-Access}) & (\varepsilon\ c\ \left\{\overline{l} = \overline{v}\right\} :: G).l & \longrightarrow \begin{cases} (\varepsilon_k \circ \varepsilon_{G'})\ u_k :: G' & l_k = l \text{ with } \varepsilon_k\ u_k :: G_k \doteq v_k \\ & \text{and } G' \doteq \widetilde{\mathsf{fty}}_{\Delta,\Xi}(l_k, G) \\ \mathbf{error_T} & \text{if } \varepsilon_k \circ \varepsilon_{G'} \text{ is not defined} \\ \mathbf{error_A} & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 10. $\lambda^{\varepsilon}_{\mathrm{D?}}$: Syntax, typing and dynamic semantics

Syntax and static semantics of $\lambda^{\varepsilon}_{\mathrm{D?}}$. The syntax of $\lambda^{\varepsilon}_{\mathrm{D?}}$ differs from that of $\lambda_{\mathrm{D?}}$ in the introduction of *evidence* in ascriptions and values (Figure 10). Intuitively, evidences justify consistent judgments locally, and during reduction, evidences are combined through a partial operation called *consistent transitivity*. If the combination is successful, reduction proceeds; otherwise a runtime type error is reported, denoting the clash between two mutually-incompatible local justifications.

The metavariable ε ranges over evidences: in a language where the consistency notion is symmetric, evidence can be represented simply by one gradual type.[4] For instance, the evidence that $? \rightarrow \mathsf{Int}$ is consistent with $\mathsf{Bool} \rightarrow ?$ is the gradual type $\mathsf{Bool} \rightarrow \mathsf{Int}$, i.e. the *meet* (relative to precision) of both types. Evidence-augmented consistency judgments are written $\varepsilon \Vdash \Delta \vdash G \sim G'$.

An ascription $\varepsilon\ e :: G$ carries the evidence that the actual type of e is consistent with G. A value v in $\lambda^{\varepsilon}_{\mathrm{D?}}$ is a *raw value* u ascribed to a gradual type, together with the supporting evidence. For convenience, we write ε_G to denote the obvious reflexive evidence that G is consistent with itself (i.e. $\varepsilon_G = G$).

Finally, $\lambda^{\varepsilon}_{\mathrm{D?}}$ introduces a new kind of runtime errors, $\mathbf{error_T}$, which correspond to runtime type errors, witnessed when reduction requires combining incompatible evidences (more below).

The typing rules of $\lambda^{\varepsilon}_{\mathrm{D?}}$ are almost identical to those of $\lambda_{\mathrm{D?}}$ (Figure 10). In the ascription rule, consistency is replaced by evidence-augmented type consistency; observe how the evidence supporting the consistency judgment is held in the ascription term itself. This is the key "runtime tracking" mechanism of AGT. Additionally, the use of consistency in other rules is replaced by type precision when the consistent judgment has one fixed gradual type. For example, in εG-Access, $\Delta \vdash G \sim ?_{\mathrm{D}}$ is replaced by $\Delta \vdash G \sqsubseteq ?_{\mathrm{D}}$.

Reduction semantics. The reduction semantics of $\lambda^{\varepsilon}_{\mathrm{D?}}$ is described in Figure 10. The ascription rule εR-AscErase, standard in evidence-based reduction semantics, is key to understand the mechanisms of runtime type checking in this technical setting. The rule describes how an ascription surrounding a value reduces to a single value if the two evidences can be combined through the *consistent transitivity* operator. Consistent transitivity is the key runtime operator in evidence-based semantics. It is the runtime type tracking mechanism, playing the dual role of type tags and casts in other presentations of gradual languages; likewise, a failure of consistent transitivity corresponds to a cast error [Garcia et al. 2016].

In our context where evidences are just gradual types, the general definition of consistent transitivity [Garcia et al. 2016] boils down to $\Delta \vdash \varepsilon_{G_1} \circ \varepsilon_{G_2} = \alpha_{\Delta}(\gamma_{\Delta}(G_1) \cap \gamma_{\Delta}(G_2))$. Note that this definition coincides with the precision meet between gradual types introduced earlier: $\Delta \vdash \varepsilon_{G_1} \circ \varepsilon_{G_2} = G_1 \sqcap G_2$.

In rule εR-AscErase, ε_1 justifies that G_u, the type of the raw value, is consistent with G_1, while ε_2 justifies $G_1 \sim G_2$. Composition of these evidences via consistent transitivity, if defined, justifies that $G_u \sim G_2$. If consistent transitivity is undefined, the reduction steps to $\mathbf{error_T}$. For example, since $\mathsf{Int} \sqcap \mathsf{Bool}$ is not defined: $\varepsilon_{\mathsf{Bool}}\ (\varepsilon_{\mathsf{Int}}\ (\varepsilon_{\mathsf{Int}}\ 1 :: \mathsf{Int}) :: ?) :: \mathsf{Bool} \longrightarrow \varepsilon_{\mathsf{Bool}}\ (\varepsilon_{\mathsf{Int}}\ 1 :: ?) :: \mathsf{Bool} \longrightarrow \mathbf{error_T}$. Crucially, consistent transitivity ensures that unclassified data does not "infiltrate" a closed datatype at runtime, because $\Delta \vdash \varepsilon_D \circ \varepsilon_{?_{\mathrm{O}}} = D \sqcap ?_{\mathrm{O}}$ is only defined if D is an *open* datatype (in which case it is equal to D).

Rule εR-Beta reduces a function application as usual if consistent transitivity between the evidence of the argument and the domain of the function's evidence is defined; otherwise it steps to $\mathbf{error_T}$. Rule εR-Delta takes the raw value from the ascription, performs the primitive operation and returns the resulting raw value wrapped in an ascription, using the codomain of the operation's evidence.

[4] In a language with (consistent) subtyping, evidences are typically represented by a pair of gradual types [Garcia et al. 2016], likewise in languages where the consistency relation is oriented [Toro et al. 2019].

The novel rules for dealing with datatypes are derived similarly. Rule εR-Match deals with pattern matching: it performs an ordinary pattern match and binding on the underlying raw value, selecting the first matching clause. Thanks to the translation function, the evidences of the discriminee's arguments will be at least as precise as the type expected by the pattern. A consistent transitivity check is not necessary. Rule εR-Access reduces successfully only if consistent transitivity between the evidence of the value and that of the expected type of the access is defined; otherwise it steps to $\mathbf{error}_{\mathbf{T}}$.

Evaluation of $\lambda_{\mathrm{D?}}$. As mentioned earlier, the evaluation of a $\lambda_{\mathrm{D?}}$ term first elaborates the term to $\lambda^{\varepsilon}_{\mathrm{D?}}$, written $\Delta; \Xi; \cdot \vdash e \rightsquigarrow e_\varepsilon : G$, and then reduces this internal term. For a $\lambda_{\mathrm{D?}}$ term e, we write $\Delta; \Xi \vdash e \Downarrow v$ (resp. $e \Downarrow \mathbf{err}$) if $\Delta; \Xi; \cdot \vdash e \rightsquigarrow e_\varepsilon : G$ and $e_\varepsilon \mapsto^* v$ (resp. $e_\varepsilon \mapsto^* \mathbf{err}$). We often write $e \Downarrow v$ instead of $\Delta; \Xi \vdash e \Downarrow v$ for brevity. The elaboration of $\lambda_{\mathrm{D?}}$ terms to $\lambda^{\varepsilon}_{\mathrm{D?}}$ terms, provided in supplementary material, is straightforward [Toro and Tanter 2020], and very similar to a standard cast insertion translation [Siek and Taha 2006]. It inserts ascriptions on every raw value and on every term whose typing derivation relies on a consistent judgment, synthesizing the corresponding evidence that supports the ascription.

To illustrate, evaluating term $(\mathsf{Foo}\ \{x = 2\}).x + 1$, where Foo is unclassified data, goes as follows:

$$
\begin{array}{lll}
 & (\mathsf{Foo}\ \{x = 2\}).x + 1 & \\
\rightsquigarrow & (\varepsilon_{\mathsf{Int}}\ (\varepsilon_{?_{\mathrm{O}}}\ \mathsf{Foo}\ \{x = \varepsilon_{\mathsf{Int}}\ (\varepsilon_{\mathsf{Int}}\ 2 :: \mathsf{Int}) :: ?\} :: ?_{\mathrm{O}}).x :: \mathsf{Int}) + (\varepsilon_{\mathsf{Int}}\ 1 :: \mathsf{Int}) & \\
\longrightarrow & (\varepsilon_{\mathsf{Int}}\ (\varepsilon_{?_{\mathrm{O}}}\ \mathsf{Foo}\ \{x = \varepsilon_{\mathsf{Int}}\ 2 :: ?\} :: ?_{\mathrm{O}}).x :: \mathsf{Int}) + (\varepsilon_{\mathsf{Int}}\ 1 :: \mathsf{Int}) & \varepsilon_{\mathsf{Int}} \circ \varepsilon_{\mathsf{Int}} = \varepsilon_{\mathsf{Int}} \\
\longrightarrow & (\varepsilon_{\mathsf{Int}}\ (\varepsilon_{\mathsf{Int}}\ 2 :: ?) :: \mathsf{Int}) + (\varepsilon_{\mathsf{Int}}\ 1 :: \mathsf{Int}) & \varepsilon_{\mathsf{Int}} \circ \varepsilon_{?} = \varepsilon_{\mathsf{Int}} \\
\longrightarrow & (\varepsilon_{\mathsf{Int}}\ 2 :: \mathsf{Int}) + (\varepsilon_{\mathsf{Int}}\ 1 :: \mathsf{Int}) & \varepsilon_{\mathsf{Int}} \circ \varepsilon_{\mathsf{Int}} = \varepsilon_{\mathsf{Int}} \\
\longrightarrow & \varepsilon_{\mathsf{Int}}\ 3 :: \mathsf{Int} &
\end{array}
$$

As another example, the transformation and evaluation of $(\mathsf{Foo}\ \{x = 2\}).x :: ?_{\mathrm{D}}$ goes as follows:

$$
\begin{array}{lll}
 & (\mathsf{Foo}\ \{x = 2\}).x :: ?_{\mathrm{D}} & \\
\rightsquigarrow & \varepsilon_{?_{\mathrm{D}}}\ (\varepsilon_{?_{\mathrm{O}}}\ \mathsf{Foo}\ \{x = \varepsilon_{\mathsf{Int}}\ (\varepsilon_{\mathsf{Int}}\ 2 :: \mathsf{Int}) :: ?\} :: ?_{\mathrm{O}}).x :: ?_{\mathrm{D}} & \\
\longrightarrow & \varepsilon_{?_{\mathrm{D}}}\ (\varepsilon_{?_{\mathrm{O}}}\ \mathsf{Foo}\ \{x = \varepsilon_{\mathsf{Int}}\ 2 :: ?\} :: ?_{\mathrm{O}}).x :: ?_{\mathrm{D}} & \varepsilon_{\mathsf{Int}} \circ \varepsilon_{\mathsf{Int}} = \varepsilon_{\mathsf{Int}} \\
\longrightarrow & \varepsilon_{?_{\mathrm{D}}}\ (\varepsilon_{\mathsf{Int}}\ 2 :: ?) :: ?_{\mathrm{D}} & \varepsilon_{\mathsf{Int}} \circ \varepsilon_{?} = \varepsilon_{\mathsf{Int}} \\
\longrightarrow & \mathbf{error}_{\mathbf{T}} & \varepsilon_{\mathsf{Int}} \circ \varepsilon_{?_{\mathrm{D}}} \text{ is not defined}
\end{array}
$$

4.4 Metatheory of $\lambda_{\mathrm{D?}}$

We now turn to the expected properties of $\lambda_{\mathrm{D?}}$ [Siek et al. 2015a]. First, $\lambda_{\mathrm{D?}}$ is type safe. Unlike type safety of λ_{D}, in general in $\lambda_{\mathrm{D?}}$ programs can fail with any error $\mathbf{err}$, irrespective of the selected matching strategy.

Theorem 4.3 (Type Safety). *If $\Delta; \Xi; \cdot \vdash e : G$, then either $e \Downarrow v$ with $\Delta; \Xi; \cdot \vdash v : G$, or $e \Downarrow \mathbf{err}$.*

The $\lambda_{\mathrm{D?}}$ type system is equivalent to the λ_{D} type system on static terms (i.e. terms without any imprecise types), using static contexts. Let $\vdash_S$ denote the typing judgment of λ_{D}.

	Typed	Declared	Unclassifed	Default	Values	Arity	Reflect	Multiple	Uniform	Exhaustive	$?_{\mathrm{D}}$
Scheme	D	struct	✓	✓	✓	✓	✓	N/A	×	×	✓
Typed Racket	G	ctor, sets	✓	✓	✓	✓	✓	✓	×	declared	✓
CDuce	G	sets	✓	✓	✓	✓	✓	✓	✓	✓	✓
OCaml	S	types	✓	✓	✓	×	×	variants	×	declared	×
Scala	S	✓	sets	✓	✓	×	✓	✓	N/A	sealed	×
Rust	S	✓	×	✓	✓	✓	×	×	N/A	✓	×
Elm	S	✓	×	✓	✓	×	×	×	N/A	✓	×
GSD	G	✓	✓	✓	NYI	NYI	NYI	$?_{\mathrm{O}}$	✓	declared	✓

Fig. 11. Language feature comparison. Languages are loosely ordered from dynamic to static, with GSD last. N/A means "not applicable"; NYI means "not yet implemented". Typed: *D*ynamic, *S*tatic, or *G*radual types. Declared: statically declared datatypes; Scheme declares `structs` but not symbols in S-expressions; OCaml's polymorphic variants are ad-hoc; others declare only constructors (ctor) or sets of types (sets). Unclassified: ad hoc constructors, like S-expressions or polymorphic variants. Default: catch-all cases. Values: match on non-constructors. Arity: match on some prefix of positional arguments or selection of record fields. Reflect: match on type of value. Multiple: constructors can belong to more than one declared type. Uniform: no distinction between declared and ad hoc constructors. Exhaustive: static checking of exhaustiveness. $?_{\mathrm{D}}$: type characterizing data.

Theorem 4.4 (Static Equivalence for Static Terms). *Let e be a static term, T a static type, and Δ and Ξ static contexts. We have $\Delta; \Xi; \cdot \vdash_S e : T$ if and only if $\Delta; \Xi; \cdot \vdash e : T$.*

Additionally, fully static terms can only fail in the same ways that the static language allows. In particular they cannot fail with a runtime type error.

Theorem 4.5 (Static Terms Do Not Fail More). *Let e be a static term, T a static type, and Δ and Ξ static contexts. If $\Delta; \Xi; \cdot \vdash e : T$ and $e \Downarrow \mathbf{err}$, then $\mathbf{err} \in \mathrm{errors}(m)$.*

Finally, $\lambda_{\mathrm{D?}}$ satisfies the gradual guarantees [Siek et al. 2015a]. First, a well-typed program remains well-typed when made less precise. (Type precision is naturally extended to terms and contexts.)

Theorem 4.6 (Static Gradual Guarantee). *If $\Delta; \Xi; \cdot \vdash e : G$ and $\Delta \vdash e \sqsubseteq e'$ then $\Delta; \Xi; \cdot \vdash e' : G$ for some G' such that $\Delta \vdash G \sqsubseteq G'$.*

Second, a program that runs without errors still does when made less precise.

Theorem 4.7 (Dynamic Gradual Guarantee). *If $\Delta; \Xi; \cdot \vdash e : G$ and $e \Downarrow v$, then for any e' such that $\Delta \vdash e \sqsubseteq e'$, we have $e' \Downarrow v'$ for some v' such that $\Delta \vdash v \sqsubseteq v'$.*

5 COMPARISON TO EXISTING LANGUAGES

We support the GSD language (Section 2) with a series of three core calculi, but core calculi alone do not a language make! In addition to the `toJSON` and `fromJSON` builtins, our GSD examples make use of a variety of extensions to pattern matching. A broad range of approaches to pattern matching already exist (Figure 11).

A brief disclaimer: we bias the listed features to those relevant to GSD; a full of analysis of data structuring mechanisms would be a serious endeavor in itself. Such extensions are critical enablers of a variety of dynamic idioms, as seen in `flatten` [Fagan 1991; Greenberg 2019].

Scheme. Scheme is the mainstream programming language most closely resembling the lambda calculus. Scheme lacks algebraic datatype definitions. The language comes with lists and a notion of symbol sufficient to encode tagged data: S-expressions are lists where the first element is a symbol (representing the name of a constructor) and possibly more values (representing arguments to that constructor). Extensions add features like record types [Kelsey 1999] and pattern matching [Godek 2020]. Even with these extensions, the standard approach for semi-structured data in Scheme is to use S-expressions.

Typed Racket. Typed Racket is a dynamic-first gradual type system for Racket, a Scheme-like language [Tobin-Hochstadt et al. 2014]. Typed Racket's static checking uses *occurrence typing* to statically guarantee type safety and exhaustive checking [Tobin-Hochstadt and Felleisen 2008]. The type system is based on union types, which accommodate unclassified data as in Scheme, i.e., tagged data as S-expressions. Typed Racket encodes open datatypes using union types and the type `Any`, comparable to ?. Constructors and datatypes are declared separately. Every constructor is in some sense a singleton type, and there is no restriction on how many types a constructor can belong to.

CDuce. CDuce is a pure functional programming language with set-theoretic types [Benzaken et al. 2003], which excels at working with unstructured data. CDuce makes no distinction between defined and undefined constructors. In fact, there is no way to statically define constructors at all! Constructors are encoded as tagged data like in Scheme. As in Typed Racket, each constructor has its own singleton type, and it can belong to multiple named datatypes. CDuce allows the same constructor name to be used with different argument types, and programmers can match against types. CDuce's set-theoretic types go well beyond Typed Racket: intersection and negation allow for *very* precise types. For an example, see Greenberg [2019].

OCaml. In addition to standard algebraic datatypes, OCaml also supports two relaxed notions of datatype: polymorphic variants [Garrigue 1998], a disciplined static approach to unclassified data; and the open `exn` type. The `exn` type is treated specially, but uses conventional (static) constructor syntax. There are two main distinctions between GSD and OCaml's polymorphic variants: (a) how polymorphic variants inhabit types and (b) how polymorphic variants can be matched. In GSD, the closest corresponding idea is $?_{\mathrm{O}}$, which represents open datatypes. There is no way in GSD to name a type of some particular bounded set of unclassified constructors, but OCaml can approximate $?_{\mathrm{O}}$ by using lower bounds ($\sim$ in Figure 11): `` [>`Foo] `` represents all types that have at least the constructor `` `Foo ``. OCaml reasons well about such bounded sets: functions can be annotated with the exact set of polymorphic variants they expect, or with upper or lower bounds; these bounds can even be inferred. In contrast to unclassified data in GSD, polymorphic variants do not mix with regular variants in a pattern match. Such a distinction can be an advantage: typos of declared constructors are caught as syntax errors by the compiler. GSD makes no such distinction, handling any constructed data in the same match expression, irrespective of whether its classified or not—allowing for a smoother transtition from a dynamic program to a more static one at the expense of some static checking.

We do not include Haskell in the table, but its affordances are similar to OCaml. Haskell lacks polymorphic variants, but features a `Dynamic` type that provides for a structured limited notion of reflection.

Scala. Scala is a statically typed programming language that combines functional and object-oriented programming [Odersky et al. 2006]. Scala's *case classes* are akin to algebraic datatypes, and can be matched on. Classes are open by default and can be extended with multiple case classes. Pattern matches on open classes cannot be checked for exhaustiveness, since case classes can be extended from outside their declaring file. But case classes marked as *sealed* can only be extended from within their source file—in this case, the compiler checks for exhaustiveness. GSD makes similar tradeoffs between allowing openness and checking exhaustiveness. While Scala does not support ad-hoc constructors, it offers great extensibility for datatypes: the programmer can extend one datatype with another; set-theoretic types like union (of arbitrary types) and intersection (of traits) cover some ad hoc use cases.

Rust. Rust is a safe systems programming language [Klabnik and Nichols 2018]. Its type system uses substructural types to enforce an ownership discipline—these powerful features have no real bearing in our setting. Rust's `enum`s resemble algebraic datatypes, and are significantly more expressive than `enum`s in C or C++. Like C and C++, though, Rust offers ways for programmers to control the representation of `enum`s, e.g., assigning them integer values. Rust pattern matching is expressive, with convenient forms for matching only parts of a datatype as well as keywords for managing memory dereference as part of pattern matches. Because of its low-level target, Rust doesn't offer much in the way of runtime reflection.

Elm. Elm is a pure functional language that compiles to JavaScript [Czaplicki and Chong 2013]. Elm takes static checking seriously: incomplete pattern matches are an error. Of the static languages considered, Elm offers the least flexibility: Elm offers no reflection of any kind; release builds forbid debug logging and exceptions; the built-in definitions and libraries force users to handle errors with option types. We do not include them in the table, but Coq, Agda, and other total languages are similar to Elm.

6 RELATED WORK

We now expand on two lines of related work: work on datatypes, and work on gradual types. Algebraic datatypes are a fixture of typed functional programming—and they are increasingly found in other languages, too [Klabnik and Nichols 2018; Odersky et al. 2006]. Several ways of extending datatypes with new variants have been explored beyond those discussed above, including in the gradual typing literature.

Zenger and Odersky [2001] define a way of modeling extensible algebraic datatypes with defaults in an object-oriented language. Datatypes are modeled as classes and every extension, with potentially multiple variants, as a subclass of the extended datatype. It is said to be "with defaults", by the way it implements pattern matching. When doing pattern matching on a subclass, if no pattern holds then the default case does a supercall. Datatypes are extensible: programmers introduce new variants using inheritance and interfaces. These extensions are said to be linear, since each extension brings only the variants of its superclass. Linearity ensures that pattern matches cannot fail at runtime. But because extensions are all statically declared, they cannot account for dynamically discovered unclassified data.

Garcia et al. [2016] develop *gradual rows*, which are rows with possible extra unknown fields. Extending our system with gradual rows would have complicated the formalism, but would have allowed for not only extensible datatypes, but constructors with gradual arity. Sekiyama and Igarashi [2020] describe a gradual language with row types and row polymorphism. They support gradual variants: like GSD and unlike polymorphic variants, there is no special syntax for static constructors. They can bound input or output variants of a function using row polymorphism [Wand 1991]. Row types are very expressive, but they come with more metatheoretical baggage. They use scoped labels [Leijen 2017], which lead to an operational semantics that does not directly correspond to an efficient implementation.

Siek and Tobin-Hochstadt [2016], Castagna and Lanvin [2017] and [Toro and Tanter 2017] describe gradual languages with union types, under different forms. Union types are a set-theoretic alternative to algebraic datatypes. As seen in the discussion of Typed Racket and CDuce (Section 5), union types are often more flexible than algebraic approaches: a single constructor can appear in multiple types, and type unions can mix primitive values like numbers or booleans with constructors. GSD can simulate some aspects of union types using the unknown datatype $?_{\mathrm{D}}$, where **match** expressions have cases for the relevant variants. GSD has no way to statically differentiate between $\mathsf{Bool} \cup \mathsf{Foo}$ and $\mathsf{Bool} \cup \mathsf{Bar}$: both are simply $?_{\mathrm{D}}$. However, $\lambda_{\mathrm{D}?}$ facilitates working with unclassified data that is explicitly not yet defined: even with negation, it is challenging to concisely express the idea of "constructor that does not appear statically in closed datatypes".

Jafery and Dunfield [2017] develop gradual datasort refinements with sum types to eliminate pattern matching errors. The range of graduality they support is much more static than what we study here: the dynamic language is ML-like, and the static language has datasort refinements, i.e., type-level reasoning about subsets of datatype constructors. Our notion of $\mathsf{valid}_{\Delta,\Xi}$ is related; it would be interesting to combine our work and theirs for a "full spectrum" system that ranges from no guarantees (Scheme-like) to some guarantees (ML-like) to strong guarantees (datasort refinements).

7 CONCLUSION

Gradual typing aims to reconcile dynamic and static approaches, but has not yet confronted a critical, defining feature of statically-typed programming languages: algebraic datatypes. After defining a simple calculus with support for nominal algebraic datatypes, we use AGT [Garcia et al. 2016] to derive $\lambda_{\mathrm{D}?}$, a core calculus for GSD, a language for gradually structured data. Our design hinges on carefully separating open and closed datatypes, and introducing two new unknown types, $?_{\mathrm{D}}$ as the ground type of data, and $?_{\mathrm{O}}$, the unknown open datatype. Gradually structured data lets programmers handle data at different levels of static precision: from ad hoc and semi-structured or "tagged" data all the way up to fully statically defined algebraic datatypes. GSD achieves all of this while using a very simple type system.

In addition to $\lambda_{D?}$'s metatheory, we implemented a corresponding language called GSD and demonstrated that GSD supports the evolution of modern data-handling applications. There is much left to be done. Our implementation of GSD is an interpreter, not a compiler, and we have not yet attempted to improve space or time performance. We build our account of name generation into primitives like `fromJSON`, and we can imagine much more robust ways of constraining the shape of unclassified data to, e.g., some particular number or pattern of arguments. Finally, real languages support interesting extensions of datatypes, like GADTs and typeclasses. Accounting for these features is important but challenging. Overall, we show that even in with simple types, gradually structured data is an expressive and flexible approach.

REFERENCES

Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. 2011. Blame for All. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2011)*. ACM Press, Austin, Texas, USA, 201–214.

Amal Ahmed, Dustin Jamner, Jeremy G. Siek, and Philip Wadler. 2017. Theorems for Free for Free: Parametricity, with and Without Types. *Proceedings of the ACM on Programming Languages* (Sept. 2017), 39:1–39:28.

Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. 2014. A Theory of Gradual Effect Systems. In *Proceedings of the 19th ACM SIGPLAN Conference on Functional Programming (ICFP 2014)*. ACM Press, Gothenburg, Sweden, 283–295.

Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. 2016. Gradual Type-and-Effect Systems. *Journal of Functional Programming* 26 (Sept. 2016), 19:1–19:69.

Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. 2003. CDuce: An XML-Centric General Purpose Language. In *Proceedings of the 8th ACM SIGPLAN Conference on Functional Programming (ICFP 2003)*. ACM Press, Uppsala, Sweden, 51–63.

Peter Buneman. 1997. Semistructured Data. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*. ACM Press, New York, NY, USA, 117–121.

Giuseppe Castagna and Victor Lanvin. 2017. Gradual Typing with Union and Intersection Types. *Proceedings of the ACM on Programming Languages* (Sept. 2017), 41:1–41:28.

Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. 2019. Gradual Typing: A New Perspective. *Proceedings of the ACM on Programming Languages* 2, POPL (Jan. 2019), 16:1–16:32.

Evan Czaplicki and Stephen Chong. 2013. Asynchronous Functional Reactive Programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*. ACM Press, Seattle, Washington, USA, 411–422.

Mike Fagan. 1991. *Soft typing: An approach to type checking for dynamically typed languages*. Ph.D. Dissertation. Rice University. https://scholarship.rice.edu/handle/1911/16439

Ronald Garcia and Matteo Cimini. 2015. Principal Type Schemes for Gradual Programs. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2015)*. ACM Press, Mumbai, India, 303–315.

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *Proceedings of the 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2016)*, Rastislav Bodík and Rupak Majumdar (Eds.). ACM Press, St Petersburg, FL, USA, 429–442. See erratum: https://www.cs.ubc.ca/ rxg/agt-erratum.pdf.

Ronald Garcia, Éric Tanter, Roger Wolff, and Jonathan Aldrich. 2014. Foundations of Typestate-Oriented Programming. *ACM Transactions on Programming Languages and Systems* 36, 4, Article 12 (Oct. 2014), 12:1–12:44 pages.

Jacques Garrigue. 1998. Programming with polymorphic variants. In *ML Workshop*, Vol. 13. Baltimore, 7.

Panicz Maciej Godek. 2020. SRFI 200: Pattern Matching. https://srfi.schemers.org/srfi-200/srfi-200.html

Michael Greenberg. 2019. The Dynamic Practice and Static Theory of Gradual Typing. In *3rd Summit on Advances in Programming Languages (SNAPL 2019) (Leibniz International Proceedings in Informatics (LIPIcs))*, Benjamin S. Lerner, Rastislav Bodík, and Shriram

Krishnamurthi (Eds.), Vol. 136. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Providence, RI, USA, 6:1–6:20.

David Herman, Aaron Tomb, and Cormac Flanagan. 2010. Space-efficient gradual typing. *Higher-Order and Sympolic Computation* 23, 2 (June 2010), 167–189.

Atsushi Igarashi, Peter Thiemann, Vasco T. Vasconcelos, and Philip Wadler. 2017b. Gradual Session Types. *Proceedings of the ACM on Programming Languages* (Sept. 2017), 38:1–38:28.

Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 2017a. On Polymorphic Gradual Typing. *Proceedings of the ACM on Programming Languages* (Sept. 2017), 40:1–40:29.

Khurram A. Jafery and Joshua Dunfield. 2017. Sums of Uncertainty: Refinements Go Gradual. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2017)*. ACM Press, Paris, France, 804–817.

Richard Kelsey. 1999. SRFI 9: Defining Record Types. https://srfi.schemers.org/srfi-9/srfi-9.html

Steve Klabnik and Carol Nichols. 2018. *The Rust Programming Language.* No Starch Press.

Nico Lehmann and Éric Tanter. 2017. Gradual Refinement Types. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2017)*. ACM Press, Paris, France, 775–788.

Daan Leijen. 2017. Type Directed Compilation of Row-Typed Algebraic Effects. In *Proceedings of the 44th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2017)*. ACM Press, Paris, France, 486–499.

Max S. New, Dustin Jamner, and Amal Ahmed. 2020. Graduality and Parametricity: Together Again for the First Time. *Proceedings of the ACM on Programming Languages* 2, POPL (Jan. 2020), 46:1–46:32.

Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. 2006. *An overview of the Scala programming language.* Technical Report LAMP-REPORT-2006-001. EPFL. https://www.scala-lang.org/docu/files/ScalaOverview.pdf

Taro Sekiyama and Atsushi Igarashi. 2020. Gradual Typing for Extensibility by Rows. (Jan. 2020). Workshop on Gradual Typing (WGT).

Ilya Sergey and Dave Clarke. 2012. Gradual Ownership Types. In *Proceedings of the 21st European Symposium on Programming Languages and Systems (ESOP 2012) (Lecture Notes in Computer Science)*, Helmut Seidl (Ed.), Vol. 7211. Springer-Verlag, Tallinn, Estonia, 579–599.

Jeremy Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Proceedings of the Scheme and Functional Programming Workshop*. 81–92.

Jeremy Siek and Walid Taha. 2007. Gradual Typing for Objects. In *Proceedings of the 21st European Conference on Object-oriented Programming (ECOOP 2007) (Lecture Notes in Computer Science)*, Erik Ernst (Ed.). Springer-Verlag, Berlin, Germany, 2–27.

Jeremy Siek and Sam Tobin-Hochstadt. 2016. The Recursive Union of Some Gradual Types. In *Wadler Festschrift (Lecture Notes in Computer Science)*, Vol. 9600. Springer-Verlag, 388–410.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015a. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages (SNAPL 2015) (Leibniz International Proceedings in Informatics (LIPIcs))*, Vol. 32. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Asilomar, California, USA, 274–293.

Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, Sam Tobin-Hochstadt, and Ronald Garcia. 2015b. Monotonic References for Efficient Gradual Typing. In *Proceedings of the 24th European Symposium on Programming Languages and Systems (ESOP 2015) (Lecture Notes in Computer Science)*, Jan Vitek (Ed.), Vol. 9032. Springer-Verlag, London, UK, 432–456.

Asumu Takikawa, T. Stephen Strickland, Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Gradual Typing for First-Class Classes. In *Proceedings of the 27th ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2012)*. ACM Press, Tucson, AZ, USA, 793–810.

Sam Tobin-Hochstadt and Matthias Felleisen. 2008. The Design and Implementation of Typed Scheme. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2008)*. ACM Press, San Francisco, CA, USA, 395–406.

Sam Tobin-Hochstadt, Vincent St-Amour, and Eric Dobson. 2014. The Typed Racket Reference. (2014).

Matías Toro, Elizabeth Labrada, and Éric Tanter. 2019. Gradual Parametricity, Revisited. *Proceedings of the ACM on Programming Languages* 2, POPL (Jan. 2019), 17:1–17:30.

Matías Toro and Éric Tanter. 2017. A Gradual Interpretation of Union Types. In *Proceedings of the 24th Static Analysis Symposium (SAS 2017) (Lecture Notes in Computer Science)*, Vol. 10422. Springer-Verlag, New York City, NY, USA, 382–404.

Matías Toro and Éric Tanter. 2020. Abstracting Gradual References. *Science of Computer Programming* 197 (Oct. 2020), 1–65.

Niki Vazou, Éric Tanter, and David Van Horn. 2018. Gradual Liquid Type Inference. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (Nov. 2018), 132:1–132:25.

Mitchell Wand. 1987. Complete Type Inference for Simple Objects. In *Proceedings of the Symposium on Logic in Computer Science (LICS 1987)*. IEEE Computer Society Press, Ithaca, New York, USA, 37–44.

Mitchell Wand. 1991. Type inference for record concatenation and multiple inheritance. *Information and Computation* 93, 1 (1991), 1–15. Selections from 1989 IEEE Symposium on Logic in Computer Science.

Roger Wolff, Ronald Garcia, Éric Tanter, and Jonathan Aldrich. 2011. Gradual Typestate. In *Proceedings of the 25th European Conference on Object-oriented Programming (ECOOP 2011) (Lecture Notes in Computer Science)*, Mira Mezini (Ed.), Vol. 6813. Springer-Verlag, Lancaster, UK, 459–483.

Matthias Zenger and Martin Odersky. 2001. Extensible Algebraic Datatypes with Defaults. In *Proceedings of the 6th ACM SIGPLAN Conference on Functional Programming (ICFP 2001)*. ACM Press, Florence, Italy, 241–252.

Appendices

Appendices

CONTENTS

Appendix A λ_D

In this section we present the complete definitions and proofs for the properties of λ_D.

A.1 Definitions

Datatypes

Datatype names	D	$\in$	DTName
Openness	O	$\in$	Open | Closed
Constructor names	c	$\in$	CtorName
Constructor sets	C	$\in$	Ctors $\subseteq$ CtorName
Labels	l	$\in$	Label

Contexts

Datatype ctxs	Δ	$::=$	$\cdot \mid \Delta, D : (C, O)$
Constructor ctxs	Ξ	$::=$	$\cdot \mid$
			$\Xi, c : (l : T) \times \cdots \times (l : T)$
Type ctxs	Γ	$::=$	$\cdot \mid \Gamma, x : T$

Types

Base types	B	$::=$	$\mathsf{Int} \mid \ldots$
Types	T	$::=$	$B \mid T \to T \mid D$

Terms

Expressions	e	$::=$	$x \mid k \mid e\ e \mid \lambda x{:}T.\ e$
		$\mid$	$e :: T \mid c\ \left\{\overline{l} = \overline{e}\right\}$
		$\mid$	$e.l \mid \mathsf{match}\ e\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\}$
Identifiers	x	$\in$	Var
Constants	k	$::=$	$0 \mid 1 \mid \ldots \mid + \mid \ldots$
Patterns	p	$::=$	$c\ \overline{x}$
Values	v	$::=$	$k \mid \lambda x{:}T.\ e \mid c\ \left\{\overline{l} = \overline{v}\right\}$
Errors	**err**	$::=$	$\mathbf{error_M} \mid \mathbf{error_A}$

Fig. 12. λ_D syntax

Well-formed Types $\boxed{\Delta \vdash T}$

$$\frac{}{\Delta \vdash B}\ \textsc{WF-Base} \qquad \frac{\vdash \Delta \qquad D \in \mathsf{dom}(\Delta)}{\Delta \vdash D}\ \textsc{WF-Datatype} \qquad \frac{\Delta \vdash T_1 \qquad \Delta \vdash T_2}{\Delta \vdash T_1 \to T_2}\ \textsc{WF-Arrow}$$

Type equality $\boxed{T = T}$

$$\frac{}{B = B}\ \textsc{Eq-Base} \qquad \frac{}{D = D}\ \textsc{Eq-Datatype} \qquad \frac{T_1 = T_1' \qquad T_2 = T_2'}{T_1 \to T_2 = T_1' \to T_2'}\ \textsc{Eq-Arrow}$$

Context well-formedness rules $\boxed{\vdash \Delta}$ $\boxed{\Delta \vdash \Xi}$ $\boxed{\Delta; \Xi \vdash \Gamma}$

$$\frac{}{\vdash \cdot}\ \Delta\textsc{-Empty} \qquad \frac{\vdash \Delta \qquad D \notin \mathsf{dom}(\Delta) \qquad \forall D' \in \mathsf{dom}(\Delta),\ \mathsf{ctors}_\Delta(D') \cap C = \emptyset}{\vdash \Delta, D : (C, O)}\ \Delta\textsc{-Ext}$$

$$\frac{\vdash \Delta}{\Delta \vdash \cdot}\ \Xi\ \textsc{-Empty} \qquad \frac{\begin{array}{c}\Delta \vdash \Xi \qquad \forall i, j. 1 \le i, j \le n, i \ne j \implies l_i \ne l_j \\ \Delta \vdash \overline{T} \qquad \overline{T'} \doteq \mathsf{fty}_{\Delta,\Xi}(\overline{l}, \mathsf{cty}_\Delta(c)) \implies \overline{T'} = \overline{T}\end{array}}{\Delta \vdash \Xi, c : (l_1 : T_1) \times \cdots \times (l_n : T_n)}\ \Xi\ \textsc{-Ext}$$

$$\frac{\Delta \vdash \Xi}{\Delta; \Xi \vdash \cdot}\ \Gamma\textsc{-Empty} \qquad \frac{\Delta; \Xi \vdash \Gamma \qquad \Delta \vdash T}{\Delta; \Xi \vdash \Gamma, x : T}\ \Gamma\textsc{-Ext}$$

Fig. 13. Type and context rules

Typing rules $\boxed{\Delta;\Xi;\Gamma \vdash e : T}$

$$\frac{\Delta;\Xi \vdash \Gamma \qquad T \doteq \Gamma(x)}{\Delta;\Xi;\Gamma \vdash x : T}\ \textsc{T-Var} \qquad \frac{}{\Delta;\Xi;\Gamma \vdash k : \mathsf{ty}(k)}\ \textsc{T-Const}$$

$$\frac{\Delta;\Xi;\Gamma \vdash e_1 : T_1 \qquad \Delta;\Xi;\Gamma \vdash e_2 : T_2 \qquad T_2 = \mathsf{dom}(T_1)}{\Delta;\Xi;\Gamma \vdash e_1\ e_2 : \mathsf{cod}(T_1)}\ \textsc{T-App}$$

$$\frac{\Delta;\Xi;\Gamma, x : T_x \vdash e : T}{\Delta;\Xi;\Gamma \vdash \lambda x{:}T_x.\ e : T_x \to T}\ \textsc{T-Lam} \qquad \frac{\Delta;\Xi;\Gamma \vdash e : T \qquad T = T'}{\Delta;\Xi;\Gamma \vdash e :: T' : T'}\ \textsc{T-Ascribe}$$

$$\frac{\begin{array}{c} T \doteq \mathsf{cty}_{\Delta}(c) \qquad \mathsf{isdata}_{\Delta}(T) \qquad \Delta;\Xi;\Gamma \vdash \overline{e} : \overline{T} \\ \overline{T} = \mathsf{lty}_{\Xi}(\overline{l}, c) \qquad \mathsf{satisfylabels}_{n\Xi}(c, l_1 \times \cdots \times l_n) \end{array}}{\Delta;\Xi;\Gamma \vdash c\ \left\{\overline{l}^{n} = \overline{e}^{n}\right\} : T}\ \textsc{T-Ctor} \qquad \frac{\Delta;\Xi;\Gamma \vdash e : T \qquad \mathsf{isdata}_{\Delta}(T)}{\Delta;\Xi;\Gamma \vdash e.l : \mathsf{fty}_{\Delta,\Xi}(l, T)}\ \textsc{T-Access}$$

$$\frac{\begin{array}{c} \Delta;\Xi;\Gamma \vdash e : T \qquad \mathsf{isdata}_{\Delta}(T) \qquad \mathsf{valid}_{\Delta,\Xi}(\{\overline{p}\}, T) \\ (\forall i.1 \le i \le n) \qquad (x_{i1} : T_{i1}) \times \cdots \times (x_{im_i} : T_{im_i}) \doteq \mathsf{parg}_{m_i\Xi}(p_i) \\ \Delta;\Xi;\Gamma, x_{i1} : T_{i1},\ \ldots,\ x_{im_i} : T_{im_i} \vdash e_i : T_i \end{array}}{\Delta;\Xi;\Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ \{\overline{p}^{n} \mapsto \overline{e}^{n}\} : \mathsf{equate}_{n\Delta}(T_1,\ \ldots,\ T_n)}\ \textsc{T-Match}$$

Reductions $\boxed{e \longrightarrow e \text{ or } \mathbf{err}}$

$$\begin{array}{lrcl}
\text{(R-Beta)} & (\lambda x{:}T.\ e)\ v & \longrightarrow & e\,[{}^{v}/_{x}] \\
\text{(R-Delta)} & k\ v & \longrightarrow & \delta(k, v) \\
\text{(R-AscErase)} & v :: T & \longrightarrow & v \\
\text{(R-Match)} & \mathsf{match}\ c\ \left\{\overline{l} = \overline{v}\right\}\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\} & \longrightarrow & \begin{cases} e_k\,[{}^{\overline{v}}/_{\overline{x_k}}] & k \text{ smallest } \mathbb{N} \text{ s.t. } c = c_k \text{ where } c_k\ \overline{x_k} \doteq p_k \\ \mathbf{error_M} & \text{there is no } k \text{ s.t. } c = c_k \end{cases} \\
\text{(R-Access)} & \left(c\ \left\{\overline{l} = \overline{v}\right\}\right).l & \longrightarrow & \begin{cases} v_k & l_k = l \\ \mathbf{error_A} & \text{otherwise} \end{cases}
\end{array}$$

Frames $\boxed{e \longmapsto e \text{ or } \mathbf{err}}$

$$E ::= \square :: T \mid \square\ e \mid v\ \square \mid \square.l \mid c\ \left\{\overline{l} = \overline{v},\ l = \square,\ \overline{l} = \overline{e}\right\} \mid \mathsf{match}\ \square\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\}$$

$$\frac{e \longrightarrow e'}{e \longmapsto e'}\ \mathrm{R}\longrightarrow \qquad \frac{e \longmapsto e'}{E[e] \longmapsto E[e']}\ \mathrm{RE}$$

$$\frac{e \longrightarrow \mathbf{err}}{e \longmapsto \mathbf{err}}\ \textsc{RErr} \qquad \frac{e \longmapsto \mathbf{err}}{E[e] \longmapsto \mathbf{err}}\ \textsc{RErr}$$

Fig. 14. λ_{D} semantics

$$
\begin{array}{lcl}
\mathsf{dom} : \mathsf{Type} \rightharpoonup \mathsf{Type} & & \\
\mathsf{dom}(T_1 \to T_2) & = & T_1 \\
\mathsf{dom}(\_) & = & \bot
\end{array}
\qquad
\begin{array}{lcl}
\mathsf{cty}_\Delta : \mathsf{Ctors} \rightharpoonup \mathsf{Type} & & \\
\mathsf{cty}_\Delta(c) & = & \bigsqcap \{ D \in \mathsf{dom}(\Delta) \ \mid \ c \in \mathsf{ctors}_\Delta(D) \}
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{cod} : \mathsf{Type} \rightharpoonup \mathsf{Type} & & \\
\mathsf{cod}(T_1 \to T_2) & = & T_2 \\
\mathsf{cod}(\_) & = & \bot
\end{array}
\qquad
\begin{array}{lcll}
\mathsf{lty}_\Xi : \mathsf{Label} \times \mathsf{Ctors} \rightharpoonup \mathsf{Type} & & & \\
\mathsf{lty}_\Xi(l, c) & = & T_i & (l_1 : T_1) \times \cdots \times (l_n : T_n) = \Xi(c) \land l = l_i \\
\mathsf{lty}_\Xi(l, c) & = & \bot & \text{otherwise}
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{ctors}_\Delta : \mathsf{Type} \rightharpoonup \mathcal{P}(\mathsf{Ctors}) & & \\
\mathsf{ctors}_\Delta(D) & = & \pi_1(\Delta(D)) \\
\mathsf{ctors}_\Delta(\_) & = & \bot
\end{array}
\qquad
\begin{array}{lcl}
\mathsf{fty}_{\Delta,\Xi} : \mathsf{Label} \times \mathsf{Type} \rightharpoonup \mathsf{Type} & & \\
\mathsf{fty}_{\Delta,\Xi}(l, D) & = & \bigsqcap \left\{ \mathsf{lty}_\Xi(l, c) \ \mid \ c \in \mathsf{ctors}_\Delta(D) \right\} \\
\mathsf{fty}_{\Delta,\Xi}(l, T) & = & \bot
\end{array}
$$

$$
\begin{array}{l}
\mathsf{parg}_{n\Xi} : \mathsf{Pattern} \rightharpoonup (\mathsf{Var} : \mathsf{Type})^n \\
\mathsf{parg}_{n\Xi}(c\ (\overline{x}^n)) \quad = \quad
\begin{cases}
(x_1 : T_1) \times \cdots \times (x_n : T_n) & (l_1 : T_1) \times \cdots \times (l_m : T_m) \doteq \Xi(c) \land n = m \\
\bot & c \notin \mathsf{dom}(\Xi) \lor n \neq m
\end{cases}
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{equate}_{n\Delta} : \mathsf{Type}^n \rightharpoonup \mathsf{Type} & & \\
\mathsf{equate}_{n\Delta}(T, \ldots, T) & = & T \\
\mathsf{equate}_{n\Delta}(\_, \ldots, \_) & = & \bot \\
& & \\
\mathsf{pctor} : \mathsf{Pattern} \to \mathsf{Ctors} & & \\
\mathsf{pctor}(c\ \overline{x}) & = & c
\end{array}
\qquad
\mathsf{valid}_{\Delta,\Xi}(P, D) \Leftrightarrow
\begin{cases}
\bigcup_{p \in P} \mathsf{pctor}(p) \subseteq \mathsf{ctors}_\Delta(D) & \textsc{Sound} \\
\bigcup_{p \in P} \mathsf{pctor}(p) = \mathsf{ctors}_\Delta(D) & \textsc{Exact} \\
\bigcup_{p \in P} \mathsf{pctor}(p) \supseteq \mathsf{ctors}_\Delta(D) & \textsc{Complete}
\end{cases}
$$

$$
\mathsf{satisfylabels}_{n\Xi}(c, l_1 \times \cdots \times l_n) \quad \Leftrightarrow \quad \mathsf{permutation}(l_1 \ldots l_n, l'_1 \ldots l'_m) \quad (l'_1 : T_1) \times \cdots \times (l'_m : T_m) \doteq \Xi(c)
$$

$$
\mathsf{isdata}_\Delta(T) \quad \Leftrightarrow \quad \exists D \in \mathsf{dom}(\Delta), D = T
\qquad\qquad
\mathsf{open}_\Delta(D) \quad \Leftrightarrow \quad \pi_2(\Delta(D)) = \mathsf{Open}
$$

Fig. 15. λ_{D}: type functions and predicates

A.2 Type Safety

Lemma A.1 (Canonical Forms). *Consider a value* $\Delta;\Xi;\cdot \vdash v : T$. *Then*

(1) If $T = B$, *then* $v = k$.
(2) If $T = T_1 \longrightarrow T_2$, *then* $v = \lambda x{:}T_1.\ e$ *or* $v = k$.
(3) If $T = D$, *then* $v = c\ \left\{\overline{l} = \overline{v}\right\}$.

Proof. By direct inspection of the formation rules of terms. □

Lemma A.2 (Substitution). *If* $\Delta;\Xi;\Gamma, x : T_x \vdash e : T$, *and* $\Delta;\Xi;\cdot \vdash v : T_x$, *then* $\Delta;\Xi;\Gamma \vdash e\,[v/x] : T$.

Proof. By induction on the derivation of $\Delta;\Xi;\Gamma, x : T_x \vdash e : T$.

Case (T-Ctor)

Then

$$\frac{T \doteq \mathsf{cty}_\Delta(c) \quad \mathsf{isdata}_\Delta(T) \quad \Delta;\Xi;\Gamma, x : T_x \vdash \overline{e} : \overline{T} \quad \overline{T} = \mathsf{lty}_\Xi(\overline{l}, c) \quad \mathsf{satisfylabels}_{n\Xi}(c, l_1 \times \cdots \times l_n)}{\Delta;\Xi;\Gamma, x : T_x \vdash c\ \left\{\overline{l} = \overline{e}\right\} : T}\ (\text{T-Ctor})$$

By induction hypothesis $\Delta;\Xi;\Gamma \vdash \overline{e}\,[v/x] : \overline{T}$.

Then $\Delta;\Xi;\Gamma \vdash c\ \left\{\overline{l} = \overline{e}\,[v/x]\right\} : T$.

Therefore $\Delta;\Xi;\Gamma \vdash c\ \left\{\overline{l} = \overline{e}\right\}[v/x] : T$ and the result holds.

Case (T-Access)

Then

$$\frac{\Delta;\Xi;\Gamma, x : T_x \vdash e_0 : T_0 \quad \mathsf{isdata}_\Delta T_0}{\Delta;\Xi;\Gamma, x : T_x \vdash e_0.l : \mathsf{fty}_{\Delta,\Xi}(l, T_0)}\ (\text{T-Access})$$

By induction hypothesis $\Delta;\Xi;\Gamma \vdash e_0\,[v/x] : T_0$.

Then $\Delta;\Xi;\Gamma \vdash e_0\,[v/x].l : \mathsf{fty}_{\Delta,\Xi}(l, T_0)$.

Therefore $\Delta;\Xi;\Gamma \vdash (e_0.l)\,[v/x] : \mathsf{fty}_{\Delta,\Xi}(l, T_0)$ and the result holds.

Case (T-Match)

Then

$$\frac{\Delta;\Xi;\Gamma, x : T_x \vdash e_0 : T_0 \quad \begin{array}{c}\mathsf{isdata}_\Delta(T_0)\\ \mathsf{valid}_{\Delta,\Xi}(\{\overline{p}\}, T_0)\end{array} \quad \begin{array}{c}\forall i.1 \le i \le n\\ (x_{i1} : T_{i1}) \times \cdots \times (x_{im_i} : T_{im_i}) \doteq \mathsf{parg}_{m_i\Xi}(p_i)\\ \Delta;\Xi;\Gamma, x : T_x, x_{i1} : T_{i1}, \ldots, x_{im_i} : T_{im_i} \vdash e_i : T\end{array}}{\Delta;\Xi;\Gamma, x : T_x \vdash \mathsf{match}\ e_0\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\} : T}\ (\text{T-Match})$$

By induction hypothesis $\Delta; \Xi; \Gamma \vdash e_0\,[^v/_x] : T_0$ and
$\Delta; \Xi; \Gamma, x_{i1} : T_{i1}, \ldots, x_{im_i} : T_{im_i} \vdash e_i\,[^v/_x] : T$.

Then $\Delta; \Xi; \Gamma \vdash \mathsf{match}\ e_0\,[^v/_x]\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\,[^v/_x]\} : T$.

Therefore $\Delta; \Xi; \Gamma \vdash (\mathsf{match}\ e_0\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\})\,[^v/_x] : T$ and the result holds.

□

Lemma A.3 (Progress). *When using matching strategy $m \in \{Sound, Exact, Complete\}$, if $\Delta; \Xi; \Gamma \vdash e : T$, then either*

- *e is a value*
- $\exists e'.\ e \longmapsto e'$
- $e \longmapsto \mathbf{err}$ *with* $\mathbf{err} \in \mathrm{errors}(m)$

Proof. By induction on the derivation of $\Delta; \Xi; \Gamma \vdash e : T$.

Case (T-Ascribe)

Then $e = e_0 :: T$ and we have

$$\frac{\Delta; \Xi; \cdot \vdash e_0 : T_0 \quad T_0 = T}{\Delta; \Xi; \cdot \vdash e_0 :: T : T}\ (\text{T-Ascribe})$$

By induction hypothesis, either e_0 is a value, $e_0 \longmapsto e_0'$, or $e_0 \longmapsto \mathbf{err}$ with $\mathbf{err} \in \mathrm{errors}(m)$.

- If e_0 is a value, then by R-AscErase and R $\longrightarrow$

$$e_0 :: T \longmapsto e_0$$

  and the result holds.

- If $e_0 \longmapsto e_0'$, then by RE

$$e_0 :: T \longmapsto e_0' :: T$$

  and the result holds.

- If $e_0 \longmapsto \mathbf{err}$ with $\mathbf{err} \in \mathrm{errors}(m)$, then by RErr

$$e_0 :: T \longmapsto \mathbf{err}$$

  and the result holds.

Case (T-Ctor)

Then $e = c\ \left\{\overline{l} = \overline{e}\right\}$ and we have

$$\frac{T \doteq \mathsf{cty}_\Delta(c) \quad \mathsf{isdata}_\Delta(T) \quad \Delta; \Xi; \cdot \vdash \overline{e} : \overline{T} \quad \overline{T} = \mathsf{lty}_\Xi(\overline{l}, c) \quad \mathsf{satisfylabels}_{n\Xi}(c, l_1 \times \cdots \times l_n)}{\Delta; \Xi; \cdot \vdash c\ \left\{\overline{l} = \overline{e}\right\} : T}\ (\text{T-Ctor})$$

By induction hypothesis, either every e_i is a value, $\exists e_k'$ s.t. $e_k \longmapsto e_k'$ and every e_i with $i < k$ is a value, or $\exists e_k'$ s.t. $e_k \longmapsto \mathbf{err}$ with $\mathbf{err} \in \mathrm{errors}(m)$ and every e_i with $i < k$ is a value.

- If every e_i is a value, then $c\ \left\{\overline{l} = \overline{e}\right\}$ is a value and the result holds.
- If $\exists e'_k$ s.t. $e_k \longmapsto e'_k$ and every e_i with $i < k$ is a value, then by RE we have
$$c\ \left\{\overline{l} = \overline{e}, l_k = e_k, \overline{l} = \overline{e}\right\} \longmapsto c\ \left\{\overline{l} = \overline{e}, l_k = e'_k, \overline{l} = \overline{e}\right\}$$
and the result holds.
- If $\exists e'_k$ s.t. $e_k \longmapsto \mathbf{err}$ with $\mathbf{err} \in \text{errors}(m)$ and every e_i with $i < k$ is a value, then by RErr we have
$$c\ \left\{\overline{l} = \overline{e}, l_k = e_k, \overline{l} = \overline{e}\right\} \longmapsto \mathbf{err}$$
and the result holds.

Case (T-Access)

Then $e = e_0.l$ and we have
$$\frac{\Delta; \Xi; \cdot \vdash e_0 : T_0 \quad \mathsf{isdata}_{\Delta}(T_0)}{\Delta; \Xi; \cdot \vdash e_0.l : \mathsf{fty}_{\Delta,\Xi}(l, T_0)}\ \text{(T-Access)}$$
Since $\text{isdata}_{\Delta}(T_0)$, then $D = T_0$ for some $D \in \mathsf{dom}(\Delta)$.

By induction hypothesis, either e_0 is a value, $e_0 \longmapsto e'_0$, or $e_0 \longmapsto \mathbf{err}$ with $\mathbf{err} \in \text{errors}(m)$.

- If e_0 is a value, then by Canonical Forms (Lemma A.1) $e_0 = c\ \left\{\overline{l} = \overline{v}\right\}$.
If $l = l_k$ for some k, then by R-Access and R $\longrightarrow$ we have
$$c\ \left\{\overline{l} = \overline{v}\right\}.l \longmapsto v_k$$
and the result holds.
If $\nexists l_k.\ l = l_k$, then by R-Access and RErr
$$c\ \left\{\overline{l} = \overline{v}\right\}.l \longmapsto \mathbf{error_A}$$
and the result also holds since $\mathbf{error_A} \in \text{errors}(m)$ for every m.
- If $e_0 \longmapsto e'_0$, then by RE
$$e_0.l \longmapsto e'_0.l$$
and the result holds.
- If $e_0 \longmapsto \mathbf{err}$ with $\mathbf{err} \in \text{errors}(m)$, then by RErr
$$e_0.l \longmapsto \mathbf{err}$$
and the result holds.

Case (T-Match)

Then $e = \mathsf{match}\ e_0\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\}$ and we have
$$\frac{\Delta; \Xi; \cdot \vdash e_0 : T_0 \quad \mathsf{isdata}_{\Delta}(T_0) \quad \mathsf{valid}_{\Delta,\Xi}(\left\{\overline{p}\right\}, T_0) \quad \begin{array}{c} \forall i.1 \le i \le n \\ (x_{i1} : T_{i1}) \times \cdots \times (x_{im_i} : T_{im_i}) \doteq \mathsf{parg}_{m_i \Xi}(p_i) \\ \Delta; \Xi; \Gamma, x_{i1} : T_{i1}, \ldots, x_{im_i} : T_{im_i} \vdash e_i : T \end{array}}{\Delta; \Xi; \cdot \vdash \mathsf{match}\ e_0\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\} : T}\ \text{(T-Match)}$$

Since $\text{isdata}_\Delta(T_0)$ we know that $D = T_0$ for some $D \in \text{dom}(\Delta)$.

By induction hypothesis, either e_0 is a value, $e_0 \longmapsto e_0'$ or $e_0 \longmapsto \textbf{err}$ with $\textbf{err} \in \text{errors}(m)$.

- If e_0 is a value, then by Canonical Forms (Lemma A.1) we have $e_0 = c\ \left\{\overline{l = v}\right\}$. Since e_0 is well-typed, then $c \in ctors(D)$.

  By an analisys of cases on m.

  - If $m = \text{Sound}$, then $\text{valid}_{\Delta,\Xi}(\left\{\overline{p}\right\}, D) \Leftrightarrow \bigcup(\text{pctor}(p_i)) \subseteq \text{ctors}_\Delta(D)$.

    If $c \in \bigcup(\text{pctor}(p_i))$, then $c = c_k$ with $c_k\overline{x_k} \doteq p_k$ for some p_k. Then by R-Match and RE

    $$\text{match } c\ \left\{\overline{l = v}\right\} \text{ with } \{\overline{p} \mapsto \overline{e}\} \longmapsto e_k\ [\overline{v}/\overline{x_k}]$$

    and the result holds.

    In contrast, if $c \notin \bigcup(\text{pctor}(p_i))$, then there is no k s.t. $c = c_k$ with $c_k\overline{x_k} \doteq p_k$. Then by Rerr

    $$\text{match } c\ \left\{\overline{l = v}\right\} \text{ with } \{\overline{p} \mapsto \overline{e}\} \longmapsto \textbf{error}_\mathbf{M}$$

    and since $\textbf{error}_\mathbf{M} \in \text{errors(Sound)}$ the result holds.

  - If $m = \text{Exact}$, then $\text{valid}_{\Delta,\Xi}(\left\{\overline{p}\right\}, D) \Leftrightarrow \bigcup(\text{pctor}(p_i)) = \text{ctors}_\Delta(D)$. Then it follows that $c \in \bigcup(\text{pctor}(p_i))$, and therefore $c = c_k$ with $c_k\overline{x_k} \doteq p_k$ for some p_k. Then by R-Match and RE

    $$\text{match } c\ \left\{\overline{l = v}\right\} \text{ with } \{\overline{p} \mapsto \overline{e}\} \longmapsto e_k\ [\overline{v}/\overline{x_k}]$$

    and the result holds.

  - If $m = \text{Complete}$, then $\text{valid}_{\Delta,\Xi}(\left\{\overline{p}\right\}, D) \Leftrightarrow \bigcup(\text{pctor}(p_i)) \supseteq \text{ctors}_\Delta(D)$. Then it follows that $c \in \bigcup(\text{pctor}(p_i))$, and therefore $c = c_k$ with $c_k\overline{x_k} \doteq p_k$ for some p_k. Then by R-Match and RE

    $$\text{match } c\ \left\{\overline{l = v}\right\} \text{ with } \{\overline{p} \mapsto \overline{e}\} \longmapsto e_k\ [\overline{v}/\overline{x_k}]$$

    and the result holds.

- If $e_0 \longmapsto e_0'$, then by RE we have

  $$\text{match } e_0 \text{ with } \{\overline{p} \mapsto \overline{e}\} \longmapsto \text{match } e_0' \text{ with } \{\overline{p} \mapsto \overline{e}\}$$

  and the result holds.

- If $e_0 \longmapsto \textbf{err}$ with $\textbf{err} \in \text{errors}(m)$

  $$\text{match } e_0 \text{ with } \{\overline{p} \mapsto \overline{e}\} \longmapsto \textbf{err}$$

  and the result holds.

□

Lemma A.4 (Preservation for $\longrightarrow$). *If $\Delta; \Xi; \cdot \vdash e : T$ and $e \longrightarrow e'$, then $\Delta; \Xi; \cdot \vdash e' : T$.*

PROOF. By induction on the derivation of $e \longrightarrow e'$.

Case (R-AscErase)

We know that $e = v :: T$. Then

$$\frac{\Delta; \Xi; \cdot \vdash v : T' \quad T' = T}{\Delta; \Xi; \cdot \vdash v :: T : T} \text{ (T-Ascribe)}$$

and by R-AscErase

$$\frac{}{v :: T \longrightarrow v} \text{ (R-AscErase)}$$

Therefore $\Delta; \Xi; \cdot \vdash v : T$, since $T' = T$, and the result holds.

Case (R-Access)

We know that $e = (c\ \left\{\bar{l} = \overline{v}\right\}).l$. Then

$$\frac{\dfrac{T' \doteq \mathsf{cty}_\Delta(c) \quad \mathsf{isdata}_\Delta(T') \quad \begin{array}{c}\Delta; \Xi; \cdot \vdash \overline{v} : \overline{T'} \quad \overline{T'} = \mathsf{lty}_\Xi(\bar{l}, c) \\ \mathsf{satisfylabels}_{n\Xi}(c, l_1 \times \cdots \times l_n)\end{array}}{\Delta; \Xi; \cdot \vdash c\ \left\{\bar{l} = \overline{v}\right\} : T'} \text{ (T-Ctor)} \quad \mathsf{isdata}_\Delta(T')}{\Delta; \Xi; \cdot \vdash (c\ \left\{\bar{l} = \overline{v}\right\}).l : \mathsf{fty}_{\Delta,\Xi}(l, T')} \text{ (T-Access)}$$

and by R-Access

$$\frac{l_k = l}{(c\ \left\{\bar{l} = \overline{v}\right\}).l \longrightarrow v_k} \text{ (R-Access)}$$

Since $\mathsf{lty}_\Xi(l, c) = \mathsf{fty}_{\Delta,\Xi}(l, \mathsf{cty}_\Delta(c))$ we have that $\Delta; \Xi; \cdot \vdash v_k : \mathsf{fty}_{\Delta,\Xi}(l, T')$ and the result holds.

Case (R-Match)

We know that $e = \mathsf{match}\ (c\ \left\{\bar{l} = \overline{v}\right\})\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\}$. Then

$$\frac{\begin{array}{c}\mathsf{isdata}_\Delta(T') \\ \mathsf{valid}_{\Delta,\Xi}(\left\{\overline{p}\right\}, T')\end{array} \quad \begin{array}{c}\forall i. 1 \le i \le n \\ (x_{i1} : T_{i1}) \times \cdots \times (x_{im_i} : T_{im_i}) \doteq \mathsf{parg}_{m_i\Xi}(p_i) \\ \Delta; \Xi; \Gamma, x_{i1} : T_{i1}, \ldots, x_{im_i} : T_{im_i} \vdash e_i : T_i\end{array} \quad \dfrac{\begin{array}{c}T' \doteq \mathsf{cty}_\Delta(c) \quad \mathsf{isdata}_\Delta(T') \\ \Delta; \Xi; \cdot \vdash \overline{v} : \overline{T'} \quad \overline{T'} = \mathsf{lty}_\Xi(\bar{l}, c) \\ \mathsf{satisfylabels}_{n\Xi}(c, l_1 \times \cdots \times l_n)\end{array}}{\Delta; \Xi; \cdot \vdash c\ \left\{\bar{l} = \overline{v}\right\} : T'} \text{ (T-Ctor)}}{\Delta; \Xi; \Gamma \vdash \mathsf{match}\ c\ \left\{\bar{l} = \overline{v}\right\}\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\} : \mathsf{equate}_{n\Delta}(T_1, \ldots, T_n)} \text{ (T-Match)}$$

and by R-Match

$$\frac{k \text{ smallest } \mathbb{N} \text{ s.t. } c = c_k \quad c_k\ \overline{x_k} \doteq p_k}{\mathsf{match}\ c\ \left\{\overline{l} = \overline{v}\right\}\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\} \longrightarrow e_k\,[\overline{v}/\overline{x_k}]}\ \text{(R-Match)}$$

By the definition of $\text{equate}_{n\Delta}$, we know that $\text{equate}_{n\Delta}(T_1, \ldots, T_k, \ldots, T_n) = T_k$.

We know that $\mathsf{lty}_{\Xi}(\bar{l}, c) = \overline{T_k}$, then by Substitution (Lemma A.2) we have $\Delta; \Xi; \cdot \vdash e_k\,[\overline{v}/\overline{x_k}] : T_k$ and the result holds.

□

Lemma A.5 (Preservation). *If $\Delta; \Xi; \cdot \vdash e : T$ and $e \longmapsto e'$, then $\Delta; \Xi; \cdot \vdash e' : T$.*

Proof. By induction on the derivation of $e \longmapsto e'$.

Case (R $\longrightarrow$) Then

$$\frac{e \longrightarrow e'}{e \longmapsto e'}\ (\text{R} \longrightarrow)$$

and the result holds immediately by lemma A.4.

Case (RE) Then $e = E[e_0]$ and we have

$$\frac{e_0 \longmapsto e_0'}{E[e_0] \longmapsto E[e_0']}\ (\text{RE})$$

Since $\Delta; \Xi; \cdot \vdash E[e_0] : T$ we know that $\Delta; \Xi; \cdot \vdash e_0 : T_0$ with $E : T_0 \to T$.

Then, by induction hypothesis we have $\Delta; \Xi; \cdot \vdash e_0' : T_0$.

Therefore $\Delta; \Xi; \cdot \vdash E[e_0'] : T$ and the result holds.

□

Theorem 3.1 (Type safety of λ_{D}). *When using matching strategy m, if $\Delta; \Xi; \cdot \vdash e : T$ then either $e \Downarrow v$ with $\Delta; \Xi; \cdot \vdash v : T$, or $e \Downarrow$ **err** where **err** $\in$ errors(m).*

Proof. Direct by Progress and Preservation (Lemmas A.3 and A.5). □

Appendix B $\lambda_{D?}$ STATIC SEMANTICS

In this section we present complete definitions, and proofs for the static semantics of $\lambda_{D?}$.

B.1 Definitions

Datatypes			
Datatype names	D	$\in$	DTName
Openness	O	$\in$	Open | Closed
Constructor names	c	$\in$	CtorName
Constructor sets	C	$\in$	Ctors $\subseteq$ CtorName
Labels	l	$\in$	Label

Types			
Base types	B	$::=$	Int | ...
Types	G	$::=$	$B \mid G \to G \mid D \mid ? \mid ?_{\mathrm{D}} \mid ?_{\mathrm{O}}$

Contexts			
Datatype contexts	Δ	$::=$	$D : (C, \mathsf{Open}) \mid \Delta, D : (C, O) \mid \Delta, ?_{\mathrm{O}} : (C, \mathsf{Open})$
Constructor contexts	Ξ	$::=$	$\cdot \mid \Xi, c : (l : G) \times \cdots \times (l : G)$
Type contexts	Γ	$::=$	$\cdot \mid \Gamma, x : G$

Terms			
Expressions	e	$::=$	$x \mid k \mid e\ e \mid \lambda x{:}G.\ e \mid e :: G \mid c\ \left\{\overline{l} = \overline{e}\right\} \mid e.l \mid \mathsf{match}\ e\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\}$
Values	v	$::=$	$k \mid \lambda x{:}G.\ e \mid c\ \left\{\overline{l} = \overline{v}\right\}$
Identifiers	x	$\in$	Var
Constants	k	$::=$	$0 \mid 1 \mid \ \ldots$
			$+ \mid \ \ldots$
Patterns	p	$::=$	$c\ \overline{x}$

Fig. 16. $\lambda_{D?}$ syntax

$$\begin{array}{rcl} \multicolumn{3}{l}{\gamma_\Delta : \mathsf{GType} \to \mathcal{P}^*(\mathsf{Type})} \\ \gamma_\Delta(B) & = & \{B\} \\ \gamma_\Delta(G_1 \to G_2) & = & \{T_1 \to T_2 \mid T_1 \in \gamma_\Delta(G_1), \\ & & \quad T_2 \in \gamma_\Delta(G_2)\} \\ \gamma_\Delta(D) & = & \{D\} \\ \gamma_\Delta(?_{\mathrm{O}}) & = & \{D \in \mathsf{dom}(\Delta) \mid \mathsf{open}_\Delta(D)\} \\ \gamma_\Delta(?_{\mathrm{D}}) & = & \{D \in \mathsf{dom}(\Delta)\} \\ \gamma_\Delta(?) & = & \mathsf{Type} \end{array}$$

$$\begin{array}{rcl} \multicolumn{3}{l}{\alpha_\Delta : \mathcal{P}^*(\mathsf{Type}) \to \mathsf{GType}} \\ \alpha_\Delta(\{B\}) & = & B \\ \alpha_\Delta\left(\left\{\overline{T_{i1} \to T_{i2}}\right\}\right) & = & \alpha_\Delta\left(\left\{\overline{T_{i1}}\right\}\right) \to \alpha_\Delta\left(\left\{\overline{T_{i2}}\right\}\right) \\ \alpha_\Delta(\{D\}) & = & D \\ \alpha_\Delta\left(\left\{\overline{D_i}\right\}\right) & = & \begin{cases} ?_{\mathrm{O}} & \forall i.\ \mathsf{open}_\Delta(D_i) \\ ?_{\mathrm{D}} & \text{otherwise} \end{cases} \\ \alpha_\Delta\left(\left\{\overline{T_i}\right\}\right) & = & ? \end{array}$$

$$\begin{array}{lcl} \multicolumn{3}{l}{\gamma : \Delta \to \mathcal{P}^*(\Delta_S)} \\ \gamma(\Delta_S) & = & \{\Delta_S\} \\ \gamma(\Delta_S,\ ?_{\mathrm{O}} : (C, \mathsf{Open}),\ \Delta'_S) & = & \{\ \Delta''_S[D \mapsto (C' \cup C'', \mathsf{Open})] \\ & & \mid\ C' \in \mathcal{P}(C),\ D : (C'', \mathsf{Open}) \in \Delta''_S(D),\ \Delta''_S \doteq \Delta_S, \Delta'_S\ \} \end{array}$$

$$\begin{array}{lcl} \multicolumn{3}{l}{\gamma : \Xi \to \mathcal{P}^*(\Xi_S)} \\ \gamma_\Delta(\cdot) & = & \emptyset \\ \gamma_\Delta(\Xi,\ c : (l_1 : G_1) \times \cdots \times (l_n : G_n)) & = & \{\ \Xi_S,\ c : (l_1 : T_1) \times \cdots \times (l_n : T_n) \\ & & \mid\ \Xi_S \in \gamma_\Delta(\Xi),\ T_1 \times \cdots \times T_n \in \gamma^n_\Delta(G_1 \times \cdots \times G_n)\ \} \end{array}$$

Fig. 17. Concretization and abstraction functions

Type Precision $\boxed{\Delta \vdash G \sqsubseteq G}$

$$\frac{}{\Delta \vdash B \sqsubseteq B}\ \textsc{P-Base} \qquad \frac{\Delta \vdash G_{11} \sqsubseteq G_{21} \qquad \Delta \vdash G_{21} \sqsubseteq G_{22}}{\Delta \vdash G_{11} \to G_{12} \sqsubseteq G_{21} \to G_{22}}\ \textsc{P-Arrow} \qquad \frac{}{\Delta \vdash D \sqsubseteq D}\ \textsc{P-Data}$$

$$\frac{}{\Delta \vdash G \sqsubseteq ?}\ \text{P-?} \qquad \frac{}{\Delta \vdash ?_{\mathrm{D}} \sqsubseteq ?_{\mathrm{D}}}\ \text{P-?}_{\mathrm{D}} \qquad \frac{}{\Delta \vdash ?_{\mathrm{O}} \sqsubseteq ?_{\mathrm{O}}}\ \text{P-?}_{\mathrm{O}}$$

$$\frac{\Delta \vdash D}{\Delta \vdash D \sqsubseteq ?_{\mathrm{D}}}\ \text{P-?}_{\mathrm{D}}\text{R} \qquad \frac{}{\Delta \vdash ?_{\mathrm{O}} \sqsubseteq ?_{\mathrm{D}}}\ \text{P-?}_{\mathrm{O}}\text{L} \qquad \frac{\Delta \vdash D \qquad \mathsf{open}_\Delta(D)}{\Delta \vdash D \sqsubseteq ?_{\mathrm{O}}}\ \text{P-?}_{\mathrm{O}}\text{R}$$

Fig. 18. Type precision

Type Consistency $\boxed{\Delta \vdash G \sim G}$

$$\frac{}{\Delta \vdash B \sim B}\ \text{C-Base} \qquad \frac{\Delta \vdash T_1 \sim T_1' \qquad \Delta \vdash T_2 \sim T_2'}{\Delta \vdash T_1 \to T_2 \sim T_1' \to T_2'}\ \text{C-Arrow} \qquad \frac{}{\Delta \vdash D \sim D}\ \text{C-Data}$$

$$\frac{}{\Delta \vdash ?_\mathrm{O} \sim ?_\mathrm{O}}\ \text{C-}?_\mathrm{O} \qquad \frac{\Delta \vdash D \qquad \mathrm{open}_\Delta(D)}{\Delta \vdash ?_\mathrm{O} \sim D}\ \text{C-}?_\mathrm{O}\text{L} \qquad \frac{\Delta \vdash D \qquad \mathrm{open}_\Delta(D)}{\Delta \vdash D \sim ?_\mathrm{O}}\ \text{C-}?_\mathrm{O}\text{R}$$

$$\frac{}{\Delta \vdash ?_\mathrm{D} \sim ?_\mathrm{D}}\ \text{C-}?_\mathrm{D} \qquad \frac{}{\Delta \vdash ?_\mathrm{D} \sim ?_\mathrm{O}}\ \text{C-}?_\mathrm{D}\text{L} \qquad \frac{}{\Delta \vdash ?_\mathrm{O} \sim ?_\mathrm{D}}\ \text{C-}?_\mathrm{D}\text{R}$$

$$\frac{\Delta \vdash D}{\Delta \vdash D \sim ?_\mathrm{D}}\ \text{C-Data L} \qquad \frac{\Delta \vdash D}{\Delta \vdash ?_\mathrm{D} \sim D}\ \text{C-Data R}$$

$$\frac{}{\Delta \vdash ? \sim G}\ \text{C-?L} \qquad \frac{}{\Delta \vdash G \sim ?}\ \text{C-?R}$$

Fig. 19. Type consistency

Meet $\boxed{\Delta \vdash G \sqcap G = G}$

$$\frac{}{\Delta \vdash B \sqcap B = B}\ \sqcap\text{-Base} \qquad \frac{}{\Delta \vdash D \sqcap D = D}\ \sqcap\text{-Data}$$

$$\frac{\Delta \vdash G_1 \sqcap G_1' = G_1'' \qquad \Delta \vdash G_2 \sqcap G_2' = G_2''}{\Delta \vdash G_1 \to G_2 \sqcap G_1' \to G_2' = G_1'' \to G_2''}\ \sqcap\text{-Arrow}$$

$$\frac{}{\Delta \vdash ?_\mathrm{O} \sqcap ?_\mathrm{O} = ?_\mathrm{O}}\ \sqcap\text{-}?_\mathrm{O} \qquad \frac{\Delta \vdash D \qquad \mathrm{open}_\Delta(D)}{\Delta \vdash ?_\mathrm{O} \sqcap D = D}\ \sqcap\text{-}?_\mathrm{O}\text{L} \qquad \frac{\Delta \vdash D \qquad \mathrm{open}_\Delta(D)}{\Delta \vdash D \sqcap ?_\mathrm{O} = D}\ \sqcap\text{-}?_\mathrm{O}\text{R}$$

$$\frac{}{\Delta \vdash ?_\mathrm{D} \sqcap ?_\mathrm{D} = ?_\mathrm{D}}\ \sqcap\text{-}?_\mathrm{D} \qquad \frac{}{\Delta \vdash ?_\mathrm{D} \sqcap ?_\mathrm{O} = ?_\mathrm{O}}\ \sqcap\text{-}?_\mathrm{D}\text{L} \qquad \frac{}{\Delta \vdash ?_\mathrm{O} \sqcap ?_\mathrm{D} = ?_\mathrm{O}}\ \sqcap\text{-}?_\mathrm{D}\text{R}$$

$$\frac{\Delta \vdash D}{\Delta \vdash D \sqcap ?_\mathrm{D} = D}\ \sqcap\text{-Data L} \qquad \frac{\Delta \vdash D}{\Delta \vdash ?_\mathrm{D} \sqcap D = D}\ \sqcap\text{-Data R}$$

$$\frac{}{\Delta \vdash ? \sqcap G = G}\ \sqcap\text{-?L} \qquad \frac{}{\Delta \vdash G \sqcap ? = G}\ \sqcap\text{-?R}$$

Fig. 20. Meet for gradual types

Typing rules $\boxed{\Delta; \Xi; \Gamma \vdash e : G}$

$$\frac{\Delta; \Xi \vdash \Gamma \qquad G \doteq \Gamma(x)}{\Delta; \Xi; \Gamma \vdash x : G}\ \textsc{G-Var} \qquad \frac{}{\Delta; \Xi; \Gamma \vdash k : \mathsf{ty}(k)}\ \textsc{G-Const}$$

$$\frac{\Delta; \Xi; \Gamma \vdash e_1 : G \qquad \Delta; \Xi; \Gamma \vdash e_2 : G' \qquad \Delta \vdash G' \sim \widetilde{\mathsf{dom}_\Delta}(G)}{\Delta; \Xi; \Gamma \vdash e_1\ e_2 : \widetilde{\mathsf{cod}_\Delta}(G)}\ \textsc{G-App}$$

$$\frac{\Delta; \Xi; \Gamma, x : G_x \vdash e : G}{\Delta; \Xi; \Gamma \vdash \lambda x{:}G_x.\ e : G_x \to G}\ \textsc{G-Lam} \qquad \frac{\Delta; \Xi; \Gamma \vdash e : G \qquad \Delta \vdash G \sim G'}{\Delta; \Xi; \Gamma \vdash e :: G' : G'}\ \textsc{G-Ascribe}$$

$$\frac{\begin{array}{c} G \doteq \widetilde{\mathsf{cty}_\Delta}(c) \qquad \Delta \vdash G \sim ?_{\mathrm{D}} \qquad \Delta; \Xi; \Gamma \vdash \overline{e} : \overline{G} \\ \Delta \vdash \overline{G} \sim \widetilde{\mathsf{lty}_\Xi}(\overline{l}, c) \qquad \mathsf{satisfylabels}_{n\Xi}(c, l_1 \times \cdots \times l_n) \end{array}}{\Delta; \Xi; \Gamma \vdash c\ \left\{ \overline{l}^n = \overline{e}^n \right\} : G}\ \textsc{G-Ctor} \qquad \frac{\Delta; \Xi; \Gamma \vdash e : G \qquad \Delta \vdash G \sim ?_{\mathrm{D}}}{\Delta; \Xi; \Gamma \vdash e.l : \widetilde{\mathsf{fty}_{\Delta,\Xi}}(l, G)}\ \textsc{G-Access}$$

$$\frac{\begin{array}{c} \Delta; \Xi; \Gamma \vdash e : G \qquad \Delta \vdash G \sim ?_{\mathrm{D}} \qquad \widetilde{\mathsf{valid}_{\Delta,\Xi}}(\{\overline{p}\}, G) \\ (\forall i. 1 \leq i \leq n) \qquad (x_{i1} : G_{i1}) \times \cdots \times (x_{im_i} : G_{im_i}) \doteq \widetilde{\mathsf{parg}_{m_i \Xi}}(p_i) \\ \Delta; \Xi; \Gamma, x_{i1} : G_{i1},\ \ldots,\ x_{im_i} : G_{im_i} \vdash e_i : G_i \end{array}}{\Delta; \Xi; \Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ \{\overline{p}^n \mapsto \overline{e}^n\} : \widetilde{\mathsf{equate}_{n\Delta}}(G_1, \ldots, G_n)}\ \textsc{G-Match}$$

Fig. 21. Static semantics for $\lambda_{\mathrm{D?}}$

$$\widetilde{\mathsf{dom}}_\Delta : \mathsf{GType} \rightharpoonup \mathsf{GType}$$
$$\begin{aligned}\widetilde{\mathsf{dom}}_\Delta(G_1 \to G_2) &= G_1\\ \widetilde{\mathsf{dom}}_\Delta(?) &= ?\\ \widetilde{\mathsf{dom}}_\Delta(\_) &= \bot\end{aligned}$$

$$\widetilde{\mathsf{cod}}_\Delta : \mathsf{GType} \rightharpoonup \mathsf{GType}$$
$$\begin{aligned}\widetilde{\mathsf{cod}}_\Delta(G_1 \to G_2) &= G_2\\ \widetilde{\mathsf{cod}}_\Delta(?) &= ?\\ \widetilde{\mathsf{cod}}_\Delta(\_) &= \bot\end{aligned}$$

$$\widetilde{\mathsf{cty}}_\Delta : \mathsf{Ctors} \rightharpoonup \mathsf{GType}$$
$$\widetilde{\mathsf{cty}}_\Delta(c) = \bigsqcap \{G \in \mathsf{dom}(\Delta) \mid c \in \mathsf{ctors}_\Delta(G)\}$$

$$\widetilde{\mathsf{equate}}_{n\Delta} : \mathsf{GType}^n \rightharpoonup \mathsf{GType}$$
$$\widetilde{\mathsf{equate}}_{n\Delta}(G_1, \ldots, G_n) = \bigsqcap \{G_1, \ldots, G_n\}$$

$$\widetilde{\mathsf{lty}}_\Xi : \mathsf{Label} \times \mathsf{Ctors} \rightharpoonup \mathsf{GType}$$
$$\widetilde{\mathsf{lty}}_\Xi(l, c) = \begin{cases} G_i & (l_1 : G_1) \times \cdots \times (l_n : G_n) \doteq \Xi(c) \wedge l = l_i \\ \bot & c \notin \mathsf{dom}(\Xi) \vee \forall i. l \neq l_i \end{cases}$$

$$\widetilde{\mathsf{fty}}_{\Delta,\Xi} : \mathsf{Label} \times \mathsf{GType} \rightharpoonup \mathsf{GType}$$
$$\begin{aligned}\widetilde{\mathsf{fty}}_{\Delta,\Xi}(l, D) &= \bigsqcap \left\{\widetilde{\mathsf{lty}}_\Xi(l, c) \mid c \in \mathsf{ctors}_\Delta(D)\right\}\\ \widetilde{\mathsf{fty}}_{\Delta,\Xi}(l, ?_{\mathrm{O}}) &= \bigsqcap \left\{\widetilde{\mathsf{lty}}_\Xi(l, c) \mid D \in \mathsf{dom}(\Delta), c \in \{\mathsf{ctors}_\Delta(D), \mathsf{ctors}_\Delta(?_{\mathrm{O}})\}, \mathsf{open}_\Delta(D)\right\}\\ \widetilde{\mathsf{fty}}_{\Delta,\Xi}(l, ?_{\mathrm{D}}) &= \bigsqcap \left\{\widetilde{\mathsf{lty}}_\Xi(l, c) \mid D \in \mathsf{dom}(\Delta), c \in \{\mathsf{ctors}_\Delta(D)\}\right\}\\ \widetilde{\mathsf{fty}}_{\Delta,\Xi}(l, ?) &= \bigsqcap \left\{\widetilde{\mathsf{lty}}_\Xi(l, c) \mid D \in \mathsf{dom}(\Delta), c \in \{\mathsf{ctors}_\Delta(D)\}\right\}\\ \widetilde{\mathsf{fty}}_{\Delta,\Xi}(l, \_) &= \bot\end{aligned}$$

$$\widetilde{\mathsf{parg}}_{n\Xi} : \mathsf{Pattern} \rightharpoonup (\mathsf{Var} : \mathsf{GType})^n$$
$$\widetilde{\mathsf{parg}}_{n\Xi}(c\ \overline{x}^n) = \begin{cases} (x_1 : G_1) \times \cdots \times (x_n : G_n) & (l_1 : G_1) \times \cdots \times (l_m : G_m) \doteq \Xi(c) \wedge n = m \\ \bot & c \notin \mathsf{dom}(\Xi) \vee n \neq m \end{cases}$$

$$\widetilde{\mathsf{valid}}_{\Delta,\Xi}(P, G) \iff \exists \Delta_S \in \gamma(\Delta),\ \exists \Xi_S \in \gamma_\Delta(\Xi),\ \exists T \in \gamma_\Delta(G),\ \mathsf{valid}_{\Delta,\Xi}(P, T)$$

$$\widetilde{\mathsf{isdata}}_\Delta(G) \iff \Delta \vdash G \sim ?_{\mathrm{D}}$$

Fig. 22. $\lambda_{\mathrm{D?}}$: consistent type functions and predicates

Type well-formedness $\boxed{\Delta \vdash G}$

$$\frac{}{\Delta \vdash B}\ \text{WFG-Base} \qquad \frac{\vdash \Delta \qquad D \in \mathsf{dom}(\Delta)}{\Delta \vdash D}\ \text{WFG-Datatype}$$

$$\frac{\Delta \vdash G_1 \qquad \Delta \vdash G_2}{\Delta \vdash G_1 \rightarrow G_2}\ \text{WFG-Arrow} \qquad \frac{\vdash \Delta}{\Delta \vdash ?}\ \text{WFG-?}$$

$$\frac{\vdash \Delta}{\Delta \vdash ?_{\mathrm{D}}}\ \text{WFG-}?_{\mathrm{D}} \qquad \frac{\vdash \Delta}{\Delta \vdash ?_{\mathrm{O}}}\ \text{WFG-}?_{\mathrm{O}}$$

Datatype context well-formedness $\boxed{\vdash \Delta}$

$$\frac{\vdash \Delta \qquad ?_{\mathrm{O}} \notin \mathsf{dom}(\Delta) \qquad \forall D' \in \mathsf{dom}(\Delta),\ \mathsf{ctors}_{\Delta}(D') \cap C = \emptyset}{\vdash \Delta, ?_{\mathrm{O}} : (C, \mathsf{Open})}\ \Delta\text{-}?_{\mathrm{O}}$$

$$\frac{}{\vdash D : (C, \mathsf{Open})}\ \Delta\text{-Empty} \qquad \frac{\vdash \Delta \qquad D \notin \mathsf{dom}(\Delta) \qquad \forall D' \in \mathsf{dom}(\Delta),\ \mathsf{ctors}_{\Delta}(D') \cap C = \emptyset}{\vdash \Delta, D : (C, O)}\ \Delta\text{-Ext}$$

Constructor context well-formedness $\boxed{\Delta \vdash \Xi}$

$$\frac{\vdash \Delta}{\Delta \vdash \cdot}\ \Xi\text{ -Empty} \qquad \frac{\begin{array}{c}\Delta \vdash \Xi \qquad \forall i, j.1 \leq i, j \leq n, i \neq j \implies l_i \neq l_j \qquad \Delta \vdash \overline{G} \\ \overline{G'} \doteq \widetilde{\mathsf{fty}}_{\Delta,\Xi}(\bar{l}, \widetilde{\mathsf{cty}}_{\Delta}(c)) \implies \Delta \vdash \overline{G} \sim \overline{G'}\end{array}}{\Delta \vdash \Xi, c : (l_1 : G_1) \times \cdots \times (l_n : G_n)}\ \Xi\text{ -Ext}$$

Type context well-formedness $\boxed{\Delta, \Xi \vdash \Gamma}$

$$\frac{\Delta \vdash \Xi}{\Delta; \Xi \vdash \cdot}\ \Gamma\text{-Empty} \qquad \frac{\Delta; \Xi \vdash \Gamma \qquad \Delta; \Xi \vdash G}{\Delta; \Xi \vdash \Gamma, x : G}\ \Gamma\text{-Ext}$$

Fig. 23. Well-formedness of types and contexts

B.2 Galois Connection

Theorem B.1 (α_Δ is Sound). *For any well-formed datatype context Δ and any non empty set of well-formed static types $S = \left\{\overline{T_i}\right\}$, we have $S \subseteq \gamma_\Delta(\alpha_\Delta(S))$*

Proof. By induction on the structure of the non empty set S.

Case $(\{D\})$ Then $\gamma_\Delta(\alpha_\Delta(\{D\})) = \gamma_\Delta(D) = \{D\}$ and the result holds.

Case $(\left\{\overline{D_i}\right\})$ We have to consider two cases:

Case $(\forall i.\ \mathsf{open}_\Delta(D_i))$

Then $\gamma_\Delta(\alpha_\Delta(\left\{\overline{D_i}\right\})) = \gamma_\Delta(?_\mathrm{O}) = \left\{D \in \mathsf{dom}(\Delta) \mid \mathsf{open}_\Delta(D)\right\}$ and because S is well-formed under Δ we have $\left\{\overline{D_i}\right\} \subseteq \left\{D \in \mathsf{dom}(D) \mid \mathsf{open}_\Delta(D)\right\}$ and the result holds.

Case **otherwise**

Then $\gamma_\Delta(\alpha_\Delta(\left\{\overline{D_i}\right\})) = \gamma_\Delta(?_\mathrm{D}) = \{D \in \mathsf{dom}(\Delta)\}$ and because S is well-formed under Δ we have $\left\{\overline{D_i}\right\} \subseteq \{D \in \mathsf{dom}(\Delta)\}$ and the result holds.

Therefore the result holds for both cases.

Case $(\left\{\overline{T_i}\right\})$ Then $\gamma_\Delta(\alpha_\Delta(\left\{\overline{T_i}\right\})) = \gamma_\Delta(?) = \mathsf{Type}$ and trivially the result holds.

□

Theorem B.2 (α_Δ is Optimal). *For any well-formed datatype context Δ and any well-formed gradual type G, we have $\Delta \vdash \alpha_\Delta(\gamma_\Delta(G)) \sqsubseteq G$.*

Proof. By induction on the structure of G.

Case (D) Then $\alpha_\Delta(\gamma_\Delta(D)) = \alpha_\Delta(\{D\}) = D$ and the result holds.

Case $(?_\mathrm{O})$ Then $\alpha_\Delta(\gamma_\Delta(?_\mathrm{O})) = \alpha_\Delta(\left\{D \in \mathsf{dom}(\Delta) \mid \mathsf{open}_\Delta(D)\right\}) = \alpha_\Delta(F)$

By cases of F

Case $(\{D\})$ Then $\alpha_\Delta(F)) = \alpha_\Delta(\{D\}) = D$ with $\mathsf{open}_\Delta(D)$. But $\Delta \vdash D \sqsubseteq ?_\mathrm{O}$ and the result holds.

Case $(\left\{\overline{D_i}\right\})$ Then $\alpha_\Delta(F) = \alpha_\Delta(\left\{\overline{D_i}\right\}) = ?_\mathrm{O}$ because $\forall i.\ \mathsf{open}_\Delta(D_i)$ and the result holds.

Case $(?_\mathrm{D})$ Then $\alpha_\Delta(\gamma_\Delta(?_\mathrm{D})) = \alpha_\Delta(\mathsf{dom}(\Delta))$

By cases of $\mathsf{dom}(\Delta)$

Case $(\{D\})$ Then $\alpha_\Delta(\mathsf{dom}(\Delta))) = \alpha_\Delta(\{D\}) = D$. But $\Delta \vdash D \sqsubseteq ?_\mathrm{D}$ and the result holds.

Case $(\left\{\overline{D_i}\right\})$ Then $\alpha_\Delta(\mathsf{dom}(\Delta)) = \alpha_\Delta(\left\{\overline{D_i}\right\}) = ?_\mathrm{D}$ and the result holds.

Case $(?)$ Then $\alpha_\Delta(\gamma_\Delta(?)) = \alpha_\Delta(\mathsf{Type}) = ?$ and the result holds.

□

THEOREM B.3 ($\langle \gamma_\Delta, \alpha_\Delta \rangle$ IS A GALOIS CONNECTION). *Let Δ be a well-formed datatype context, S a non empty set of static types, and G a well-formed gradual type: (a) $S \subseteq \gamma_\Delta(\alpha_\Delta(S))$ and (b) $\Delta \vdash \alpha_\Delta(\gamma_\Delta(G)) \sqsubseteq G$.*

PROOF. By theorems B.1 and B.2. □

B.3 Static Equivalence For Static Terms

LEMMA B.4 (STATIC EQUIVALENCE FOR STATIC OPEN TERMS). *Let Δ, Ξ and Γ be static contexts, e a static term and G a static type ($G = T$). We have $\Delta; \Xi; \Gamma \vdash_S e : T$ if and only if $\Delta; \Xi; \Gamma \vdash e : T$.*

PROOF. The proof is direct thanks to the equivalence between the typing rules and the equivalence between type equality and consistency for the static types. We only present one case to illustrate the reasoning.

First, we prove $\Delta; \Xi; \Gamma \vdash_S e : T \implies \Delta; \Xi; \Gamma \vdash e : T$ by induction on the judgment $\Delta; \Xi; \Gamma \vdash_S e : T$.

Case ($\Delta; \Xi; \Gamma \vdash_S c \left\{\bar{l} = \bar{e}\right\} : T$)

Then $e = c \left\{\bar{l} = \bar{e}\right\}$, and we have

$$\frac{T \doteq \mathsf{cty}_\Delta(c) \quad \mathsf{isdata}_\Delta(T) \quad \Delta; \Xi; \Gamma \vdash_S \bar{e} : \overline{T} \quad \overline{T} = \mathsf{lty}_\Xi(\bar{l}, c) \quad \mathsf{satisfylabels}_{n\Xi}(c, l_1 \times \cdots \times l_n)}{\Delta; \Xi; \Gamma \vdash_S c \left\{\bar{l} = \bar{e}\right\} : T} \text{(T-CTOR)}$$

By induction hypothesis $\Delta; \Xi; \Gamma \vdash \bar{e} : \overline{T}$.

Then $\Delta \vdash T \sim ?_\mathrm{D}$ holds since $\mathsf{isdata}_\Delta(T)$ holds, and $\widetilde{\mathsf{cty}_\Delta}(c) = \mathsf{cty}_\Delta(c)$, since Δ is static, and $\widetilde{\mathsf{lty}_\Xi}(\bar{l}, c) = \mathsf{lty}_\Xi(\bar{l}, c)$, since Ξ is static.

Finally every equality judgment can be lifted into a consistency judgment.

Therefore $\Delta; \Xi; \Gamma \vdash c \left\{\bar{l} = \bar{e}\right\} : T$ and the result holds.

Then, we prove $\Delta; \Xi; \Gamma \vdash e : T \implies \Delta; \Xi; \Gamma \vdash_S e : T$ by induction on the judgment $\Delta; \Xi; \Gamma \vdash e : T$.

Case ($\Delta; \Xi; \Gamma \vdash c \left\{\bar{l} = \bar{e}\right\} : T$)

Then $e = c \left\{\bar{l} = \bar{e}\right\}$, and we have

$$\frac{T \doteq \widetilde{\mathsf{cty}_\Delta}(c) \quad \Delta \vdash T \sim ?_\mathrm{D} \quad \Delta; \Xi; \Gamma \vdash \bar{e} : \overline{T} \quad \Delta \vdash \overline{T} \sim \widetilde{\mathsf{lty}_\Xi}(\bar{l}, c) \quad \mathsf{satisfylabels}_{n\Xi}(c, l_1 \times \cdots \times l_n)}{\Delta; \Xi; \Gamma \vdash c \left\{\bar{l} = \bar{e}\right\} : T} \text{(G-CTOR)}$$

By induction hypothesis $\Delta; \Xi; \Gamma \vdash_S \bar{e} : \overline{T}$.

Then $\mathsf{isdata}_\Delta(T)$, since T is static and $\Delta \vdash T \sim ?_\mathrm{D}$, and $\mathsf{cty}_\Delta(c) = \widetilde{\mathsf{cty}_\Delta}(c)$ because Δ is static, and $\widetilde{\mathsf{lty}_\Xi}(\bar{l}, c) = \mathsf{lty}_\Xi(\bar{l}, c)$, since Ξ is static.

Finally, since all types are static, the needed consistency judgments can be replaced by equality judgments.

Therefore $\Delta;\Xi;\Gamma \vdash_S c\ \left\{\bar{l} = \overline{e}\right\} : T$ and the result holds.

□

Theorem 4.4 (Static Equivalence for Static Terms). *Let e be a static term, T a static type, and Δ and Ξ static contexts. We have $\Delta;\Xi;\cdot \vdash_S e : T$ if and only if $\Delta;\Xi;\cdot \vdash e : T$.*

Proof. Direct from lemma B.4 □

B.4 Static Gradual Guarantee

In this section we present the proof for the *static gradual guarantee*. In definitions B.5 to B.8 we present term and context precision.

Definition B.5 (Term precision).

$$\frac{}{\Delta \vdash x \sqsubseteq x}\ \text{P-Var} \qquad \frac{}{\Delta \vdash k \sqsubseteq k}\ \text{P-Const}$$

$$\frac{\Delta \vdash e \sqsubseteq e' \qquad \Delta \vdash G \sqsubseteq G'}{\Delta \vdash \lambda x{:}G.\ e \sqsubseteq \lambda x{:}G'.\ e'}\ \text{P-Lam} \qquad \frac{\Delta \vdash e \sqsubseteq e' \qquad \Delta \vdash G \sqsubseteq G'}{\Delta \vdash e :: G \sqsubseteq e' :: G'}\ \text{P-Asc}$$

$$\frac{\Delta \vdash e_1 \sqsubseteq e_1' \qquad \Delta \vdash e_2 \sqsubseteq e_2'}{\Delta \vdash e_1\ e_2 \sqsubseteq e_1'\ e_2'}\ \text{P-App} \qquad \frac{\Delta \vdash \overline{e} \sqsubseteq \overline{e'}}{\Delta \vdash c\ \left\{\bar{l} = \overline{e}\right\} \sqsubseteq c\ \left\{\bar{l} = \overline{e'}\right\}}\ \text{P-Ctor}$$

$$\frac{\Delta \vdash e \sqsubseteq e'}{\Delta \vdash e.l \sqsubseteq e'.l}\ \text{P-Access} \qquad \frac{\Delta \vdash e_0 \sqsubseteq e_0' \qquad \Delta \vdash \overline{e} \sqsubseteq \overline{e'}}{\Delta \vdash \mathsf{match}\ e_0\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\} \sqsubseteq \mathsf{match}\ e_0'\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e'}\}}\ \text{P-Match}$$

Definition B.6 (Datatype context precision).

$$\frac{}{\Delta_S \sqsubseteq \Delta_S}\ \text{P-}\Delta\ \text{Empty} \qquad \frac{C \subseteq C'}{\Delta \vdash \Delta_S, ?_{\mathrm{O}} : (C, \mathsf{Open}), \Delta_S' \sqsubseteq \Delta_S, ?_{\mathrm{O}} : (C', \mathsf{Open}), \Delta_S'}\ \text{P-}\Delta\ \text{Ext}$$

Definition B.7 (Constructor context precision).

$$\frac{\vdash \Delta}{\Delta \vdash \cdot \sqsubseteq \cdot}\ \text{P-}\Xi\ \text{Empty} \qquad \frac{\Delta \vdash \Xi \sqsubseteq \Xi' \qquad \Delta \vdash \overline{G} \sqsubseteq \overline{G'}}{\Delta \vdash \Xi, c : (l_1 : G_1) \times \cdots \times (l_n : G_n) \sqsubseteq \Xi', c : (l_1 : G_1') \times \cdots \times (l_n : G_n')}\ \text{P-}\Xi\ \text{Ext}$$

Definition B.8 (Type context precision).

$$\frac{\vdash \Delta}{\Delta \vdash \cdot \sqsubseteq \cdot}\ \text{P-EmptyCtx} \qquad \frac{\Delta \vdash \Gamma \sqsubseteq \Gamma' \qquad \Delta \vdash G \sqsubseteq G'}{\Delta \vdash \Gamma, x : G \sqsubseteq \Gamma', x : G'}\ \text{P-ExtCtx}$$

Lemma B.9. *If $\Delta \vdash G_1 \sim G_2$, $\Delta \vdash G_1 \sqsubseteq G_1'$ and $\Delta \vdash G_2 \sqsubseteq G_2'$ then $\Delta \vdash G_1' \sim G_2'$.*

Proof. By definition of type consistency, there exists T_1 and T_2 such that $T_1 \in \gamma_\Delta(G_1)$, $T_2 \in \gamma_\Delta(G_2)$ and $T_1 = T_2$. $\Delta \vdash G_1 \sqsubseteq G_1'$ and $\Delta \vdash G_2 \sqsubseteq G_2'$ mean that $\gamma_\Delta(G_1) \subseteq \gamma_\Delta(G_1')$ and $\gamma_\Delta(G_2) \subseteq \gamma_\Delta(G_2')$. Then $T_1 \in \gamma_\Delta(G_1')$ and $T_2 \in \gamma_\Delta(G_2')$. Therefore $\Delta \vdash G_1' \sim G_2'$ and the result holds. □

Theorem B.10 (Lifting of isdata_Δ). *$\Delta \vdash G \sim ?_{\mathrm{D}}$ is the natural lifting of $\mathsf{isdata}_\Delta(G)$ under γ_Δ.*

Proof. The natural lifting of isdata_Δ is defined as

$$\begin{array}{rcl}
\exists T \in \gamma_\Delta(G),\ \exists \Delta_S \in \gamma(\Delta),\ \mathsf{isdata}_\Delta(T) & \Leftrightarrow & \exists T \in \gamma_\Delta(G),\ \exists \Delta_S \in \gamma(\Delta),\ \exists D \in \mathsf{dom}(\Delta_S),\ \Delta_S \vdash D = T \\
& \Leftrightarrow & \exists T \in \gamma_\Delta(G),\ \exists \Delta_S \in \gamma(\Delta),\ \exists D \in \gamma_\Delta(?_{\mathrm{D}}),\ \Delta_S \vdash D = T \\
& \Leftrightarrow & \Delta \vdash G \sim ?_{\mathrm{D}}
\end{array}$$

and the result holds. □

Lemma B.11. *If $\Delta \vdash G \sqsubseteq G'$ and both $\widetilde{\mathsf{dom}}_\Delta(G)$ and $\widetilde{\mathsf{dom}}_\Delta(G')$ are defined then $\Delta \vdash \widetilde{\mathsf{dom}}_\Delta(G) \sqsubseteq \widetilde{\mathsf{dom}}_\Delta(G')$.*

Proof. By induction on the derivation of $\Delta \vdash G \sqsubseteq G'$.

Case $(\Delta \vdash G_1 \longrightarrow G_2 \sqsubseteq G_1' \longrightarrow G_2')$

Then we know that $\Delta \vdash G_1 \sqsubseteq G_1'$, $\widetilde{\mathsf{dom}}_\Delta(G) = G_1$ and $\widetilde{\mathsf{dom}}_\Delta(G') = G_1'$. And the result holds.

Case $(\Delta \vdash G_1 \longrightarrow G_2 \sqsubseteq ?)$

Then we know that $\widetilde{\mathsf{dom}}_\Delta(G) = G_1$ and $\widetilde{\mathsf{dom}}_\Delta(G') = ?$ and the result holds because $\Delta \vdash G_1 \sqsubseteq ?$.

Case $(\Delta \vdash ? \sqsubseteq ?)$

The we know that $\widetilde{\mathsf{dom}}_\Delta(G) = ?$ and $\widetilde{\mathsf{dom}}_\Delta(G') = ?$ and the result holds because $\Delta \vdash ? \sqsubseteq ?$.

Case $(\Delta \vdash G \sqsubseteq G')$

$\widetilde{\mathsf{dom}}_\Delta(G)$ is not defined in the rest of the cases so the result holds trivially.

□

Lemma B.12. *If $\Delta \vdash G \sqsubseteq G'$ and both $\widetilde{\mathsf{cod}}_\Delta(G)$ and $\widetilde{\mathsf{cod}}_\Delta(G')$ are defined then $\Delta \vdash \widetilde{\mathsf{cod}}_\Delta(G) \sqsubseteq \widetilde{\mathsf{cod}}_\Delta(G')$.*

Proof. Analogous to lemma B.11. □

Lemma B.13. *If $\Delta \vdash G \sqsubseteq G'$ and $\widetilde{\mathsf{fty}}_{\Delta,\Xi}(l, G)$ is defined then $\Delta \vdash \widetilde{\mathsf{fty}}_{\Delta,\Xi}(l, G) \sqsubseteq \widetilde{\mathsf{fty}}_{\Delta,\Xi}(l, G')$.*

Proof. Because $\Delta \vdash G \sqsubseteq G'$ we have $\gamma_\Delta(G) \subseteq \gamma_\Delta(G')$.

Let

$$\begin{array}{rcl}
S & = & \left\{\mathsf{fty}_{\Delta,\Xi}(l, T) \ \middle|\ T \in \gamma_\Delta(G), \Xi_S \in \gamma_\Delta(\Xi), \Delta_S \in \gamma(\Delta)\right\} \\
S' & = & \left\{\mathsf{fty}_{\Delta,\Xi}(l, T') \ \middle|\ T' \in \gamma_\Delta(G'), \Xi_S \in \gamma_\Delta(\Xi), \Delta_S \in \gamma(\Delta)\right\}
\end{array}$$

Then $S \subseteq S'$ and therefore $\Delta \vdash \alpha_\Delta(S) \sqsubseteq \alpha_\Delta(S')$.

Finally $\Delta \vdash \widetilde{\mathsf{fty}}_{\Delta,\Xi}(l, G) \sqsubseteq \widetilde{\mathsf{fty}}_{\Delta,\Xi}(l, G')$ and the result holds. □

Lemma B.14. *If $\Delta \vdash G_i \sqsubseteq G_i'$ for $i \in [1, n]$ and $\widetilde{\mathsf{equate}}_{n\Delta}(G_1, \ \ldots, G_n)$ is defined, then $\Delta \vdash \widetilde{\mathsf{equate}}_{n\Delta}(G_1, \ \ldots, G_n) \sqsubseteq \widetilde{\mathsf{equate}}_{n\Delta}(G_1', \ \ldots, G_n')$.*

Proof. Let

$$
\begin{aligned}
S &= \left\{\mathrm{equate}_{n\Delta}(T_1, \ldots, T_n) \;\middle|\; T_1 \times \cdots \times T_n \in \gamma^n_\Delta(G_1 \times \cdots \times G_n), \Delta_S \in \gamma(\Delta)\right\} \\
S' &= \left\{\mathrm{equate}_{n\Delta}(T_1, \ldots, T_n) \;\middle|\; T_1 \times \cdots \times T_n \in \gamma^n_\Delta(G'_1 \times \cdots \times G'_n), \Delta_S \in \gamma(\Delta)\right\}
\end{aligned}
$$

By definition of $\Delta \vdash G_i \sqsubseteq G'_i$, $\gamma_\Delta(G_i) \subseteq \gamma_\Delta(G'_i)$.

Then $S \subseteq S'$, and therefore $\Delta \vdash \alpha_\Delta(S) \sqsubseteq \alpha_\Delta(S')$.

Finally $\Delta \vdash \widetilde{\mathrm{equate}}_{n\Delta}(G_1,\ \ldots, G_n) \sqsubseteq \widetilde{\mathrm{equate}}_{n\Delta}(G'_1,\ \ldots, G'_n)$ and the result holds. □

Lemma B.15. *If* $\widetilde{\mathrm{valid}}_{\Delta,\Xi}(P, G)$ *and* $\Delta \vdash G \sqsubseteq G'$ *then* $\widetilde{\mathrm{valid}}_{\Delta,\Xi}(P, G')$.

Proof. By definition of $\widetilde{\mathrm{valid}}_{\Delta,\Xi}$, we have $\exists \Delta_S \in \gamma(\Delta),\ \exists \Xi_S \in \gamma_\Delta(\Xi),\ \exists T \in \gamma_\Delta(G),\ \mathrm{valid}_{\Delta,\Xi}(P, T)$.

Since $\Delta \vdash G \sqsubseteq G'$, we have $\gamma_\Delta(G) \subseteq \gamma_\Delta(G')$ and then $T \in \gamma_\Delta(G')$.

Then $\exists \Delta_S \in \gamma(\Delta),\ \exists \Xi_S \in \gamma_\Delta(\Xi),\ \exists T \in \gamma_\Delta(G'),\ \mathrm{valid}_{\Delta,\Xi}(P, T)$.

Therefore $\widetilde{\mathrm{valid}}_{\Delta,\Xi}(P, G')$, and the result holds. □

Lemma B.16. *If* $\Delta; \Xi; \Gamma \vdash e : G$ *and* $\Delta \vdash \Gamma \sqsubseteq \Gamma'$ *, then* $\Delta; \Xi; \Gamma' \vdash e : G'$ *for some* G' *such that* $\Delta \vdash G \sqsubseteq G'$.

Proof. By induction on type derivation $\Delta; \Xi; \Gamma \vdash e : G$. Only interesting cases are shown.

Case (G-Ascribe)

Then we know that

$$
\frac{\Delta; \Xi; \Gamma \vdash e_0 : G_0 \quad \Delta \vdash G_0 \sim G}{\Delta; \Xi; \Gamma \vdash e_0 :: G : G}\ (\text{G-Ascribe})
$$

By induction hypothesis $\Delta; \Xi; \Gamma' \vdash e_0 : G'_0$ with $\Delta \vdash G_0 \sqsubseteq G'_0$.

Using lemma B.9 and $\Delta \vdash G \sqsubseteq G$ we have $\Delta \vdash G'_0 \sim G$.

Therefore we get $\Delta; \Xi; \Gamma' \vdash e_0 :: G : G$ and the result holds.

Case (G-Ctor)

Then we know that

$$
\frac{G \doteq \widetilde{\mathrm{cty}}_\Delta(c) \quad \Delta \vdash G \sim ?_{\mathrm{D}} \quad \Delta; \Xi; \Gamma \vdash \overline{e} : \overline{G} \quad \Delta \vdash \overline{G} \sim \widetilde{\mathrm{lty}}_\Xi(\overline{l}, c) \quad \mathrm{satisfylabels}_{n\Xi}(c, l_1 \times \cdots \times l_n)}{\Delta; \Xi; \Gamma \vdash c\ \left\{\overline{l} = \overline{e}\right\} : G}\ (\text{G-Ctor})
$$

By induction hypothesis $\Delta; \Xi; \Gamma' \vdash \overline{e} : \overline{G'}$ with $\Delta \vdash \overline{G} \sqsubseteq \overline{G'}$.

Then $\Delta \vdash \overline{G'} \sim \widetilde{\mathrm{lty}}_\Xi(\overline{l}, c)$ by lemma B.9.

Therefore $\Delta; \Xi; \Gamma' \vdash c\ \left\{\overline{l} = \overline{e}\right\} : G$ and the result holds.

Case (G-Access)

We know that

$$\frac{\Delta;\Xi;\Gamma \vdash e_0 : G_0 \quad \Delta \vdash G_0 \sim ?_\mathrm{D}}{\Delta;\Xi;\Gamma \vdash e_0.l : \widetilde{\mathsf{fty}}_{\Delta,\Xi}(l, G_0)} \text{ (G-Access)}$$

By induction hypothesis $\Delta;\Xi;\Gamma' \vdash e_0 : G_0'$ with $\Delta \vdash G_0 \sqsubseteq G_0'$.

Then $\Delta \vdash \widetilde{\mathsf{fty}}_{\Delta,\Xi}(l, G_0) \sqsubseteq \widetilde{\mathsf{fty}}_{\Delta,\Xi}(l, G_0')$ by lemma B.13 and $\Delta \vdash G_0' \sim ?_\mathrm{D}$ by lemma B.9.

Therefore $\Delta;\Xi;\Gamma' \vdash e_0.l : \widetilde{\mathsf{fty}}_{\Delta,\Xi}(l, G_0')$ and the result holds.

Case (G-Match)

We know that

$$\frac{\Delta;\Xi;\Gamma \vdash e_0 : G_0 \quad \Delta \vdash G_0 \sim ?_\mathrm{D} \quad \widetilde{\mathsf{valid}}_{\Delta,\Xi}(\{\overline{p}\}, G_0) \quad \begin{array}{c}\forall i. 1 \leq i \leq n \\ (x_{i1} : G_{i1}) \times \cdots \times (x_{im_i} : G_{im_i}) \doteq \widetilde{\mathsf{parg}}_{m_i \Xi}(p_i) \\ \Delta;\Xi;\Gamma, x_{i1} : G_{i1}, \ldots, x_{im_i} : G_{im_i} \vdash e_i : G_i\end{array}}{\Delta;\Xi;\Gamma \vdash \mathsf{match}\ e_0\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\} : \widetilde{\mathsf{equate}}_{n\Delta}(G_1, \ldots, G_n)} \text{ (G-Match)}$$

By induction hypothesis $\Delta;\Xi;\Gamma' \vdash e_0 : G_0'$ with $\Delta \vdash G_0 \sqsubseteq G_0'$ and
$\Delta;\Xi;\Gamma', x_{i1} : G_{i1}', \ldots, x_{im_i} : G_{im_i}' \vdash e_i : G_i'$ with $\Delta \vdash G_i \sqsubseteq G_i'$ and
$\Delta \vdash G_{i1} \sqsubseteq G_{i1}', \ldots, \Delta \vdash G_{im_i} \sqsubseteq G_{im_i}'$ for $i \in [1, n]$.

Then $\Delta \vdash G_0' \sim ?_\mathrm{D}$ by lemma B.9.

And $\widetilde{\mathsf{valid}}_{\Delta,\Xi}(\{\overline{p}\}, G_0')$ by lemma B.15.

Also $\Delta \vdash \widetilde{\mathsf{equate}}_{n\Delta}(G_1, \ldots, G_n) \sqsubseteq \widetilde{\mathsf{equate}}_{n\Delta}(G_1', \ldots, G_n')$ by lemma B.14.

Therefore $\Delta;\Xi;\Gamma' \vdash \mathsf{match}\ e_0\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\} : \widetilde{\mathsf{equate}}_{n\Delta}(G_1', \ldots, G_n')$ and the result holds.

□

Lemma B.17. *If* $\Delta \vdash e \sqsubseteq e'$ *and* $\Delta \vdash v \sqsubseteq v'$, *then* $\Delta \vdash e\,[v/x] \sqsubseteq e'\,[v'/x]$.

Proof. By induction on the derivation of $\Delta \vdash e \sqsubseteq e'$. Only one case is shown, since all cases follow the same structure.

Case (P-Ctor)

Then

$$\frac{\Delta \vdash \overline{e} \sqsubseteq \overline{e'}}{\Delta \vdash c\ \left\{\overline{l} = \overline{e}\right\} \sqsubseteq c\ \left\{\overline{l} = \overline{e'}\right\}} \text{ (P-Ctor)}$$

By induction hypothesis $\Delta \vdash \overline{e}\,[v/x] \sqsubseteq \overline{e'}\,[v'/x]$.

Then $\Delta \vdash c\ \left\{\overline{l} = \overline{e}\,[v/x]\right\} \sqsubseteq c\ \left\{\overline{l} = \overline{e'}\,[v'/x]\right\}$ by P-Ctor.

Therefore $\Delta \vdash c\ \left\{\overline{l} = \overline{e}\right\}[v/x] \sqsubseteq c\ \left\{\overline{l} = \overline{e'}\right\}[v'/x]$ and the result holds.

□

THEOREM B.18 (STATIC GRADUAL GUARANTEE FOR OPEN TERMS). *If* $\Delta; \Xi; \Gamma \vdash e : G$ *and* $\Delta \vdash e \sqsubseteq e'$ *then* $\Delta; \Xi; \Gamma \vdash e' : G'$ *for some* G' *such that* $\Delta \vdash G \sqsubseteq G'$.

PROOF. Suppose $\Delta; \Xi; \Gamma \vdash e : G$ and $\Delta \vdash e \sqsubseteq e'$. By induction on the derivation of $\Delta \vdash e \sqsubseteq e'$.

Only interesting cases are shown.

Case (P-Ctor)

We have

$$\frac{\Delta \vdash \overline{e} \sqsubseteq \overline{e'}}{\Delta \vdash c\ \left\{\overline{l} = \overline{e}\right\} \sqsubseteq c\ \left\{\overline{l} = \overline{e'}\right\}}\ (\text{P-CTOR})$$

And by typing inversion on G-Ctor

$$\frac{G \doteq \widetilde{\mathsf{cty}_\Delta}(c) \quad \Delta \vdash G \sim ?_{\mathrm{D}} \quad \Delta; \Xi; \Gamma \vdash \overline{e} : \overline{G} \quad \Delta \vdash \overline{G} \sim \widetilde{\mathsf{lty}_\Xi}(\overline{l}, c) \quad \mathsf{satisfylabels}_{n\Xi}(c, l_1 \times \cdots \times l_n)}{\Delta; \Xi; \Gamma \vdash c\ \left\{\overline{l} = \overline{e}\right\} : G}\ (\text{G-CTOR})$$

By induction hypothesis we have $\Delta; \Xi; \Gamma \vdash \overline{e'} : \overline{G'}$ with $\Delta \vdash \overline{G} \sqsubseteq \overline{G'}$.

Then $\Delta \vdash \overline{G'} \sim \widetilde{\mathsf{lty}_\Xi}(\overline{l}, c)$ by lemma B.9.

Therefore $\Delta; \Xi; \Gamma \vdash c\ \left\{\overline{l} = \overline{e'}\right\} : G$ and the result holds.

Case (P-Access)

Then we have

$$\frac{\Delta \vdash e_0 \sqsubseteq e_0'}{\Delta \vdash e_0.l \sqsubseteq e_0'.l}\ (\text{P-ACCESS})$$

And by typing inversion on G-Access

$$\frac{\Delta; \Xi; \Gamma \vdash e_0 : G_0 \quad \Delta \vdash G_0 \sim ?_{\mathrm{D}}}{\Delta; \Xi; \Gamma \vdash e_0.l : \widetilde{\mathsf{fty}_{\Delta,\Xi}}(l, G_0)}\ (\text{G-ACCESS})$$

By induction hypothesis $\Delta; \Xi; \Gamma \vdash e_0' : G_0'$ with $\Delta \vdash G_0 \sqsubseteq G_0'$.

Then $\Delta \vdash \widetilde{\mathsf{fty}_{\Delta,\Xi}}(l, G_0) \sqsubseteq \widetilde{\mathsf{fty}_{\Delta,\Xi}}(l, G_0')$ by lemma B.13 and $\Delta \vdash G_0' \sim ?_{\mathrm{D}}$ by lemma B.9.

Therefore $\Delta; \Xi; \Gamma \vdash e_0'.(l) : \widetilde{\mathsf{fty}_{\Delta,\Xi}}(l, G_0')$ and the result holds.

Case (P-Match)

Then we have

$$\frac{\Delta \vdash e_0 \sqsubseteq e_0' \quad \Delta \vdash \overline{e} \sqsubseteq \overline{e'}}{\Delta \vdash \mathsf{match}\ e_0\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\} \sqsubseteq \mathsf{match}\ e_0'\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e'}\}}\ (\text{P-Match})$$

And by typing inversion on G-Match

$$\frac{\Delta; \Xi; \Gamma \vdash e_0 : G_0 \quad \Delta \vdash G_0 \sim ?_{\mathsf{D}} \quad \widetilde{\mathsf{valid}}_{\Delta,\Xi}(\{\overline{p}\}, G_0) \quad \begin{array}{c} \forall i.1 \leq i \leq n \\ (x_{i1} : G_{i1}) \times \cdots \times (x_{im_i} : G_{im_i}) \doteq \widetilde{\mathsf{parg}}_{m_i \Xi}(p_i) \\ \Delta; \Xi; \Gamma, x_{i1} : G_{i1}, \ldots, x_{im_i} : G_{im_i} \vdash e_i : G_i \end{array}}{\Delta; \Xi; \Gamma \vdash \mathsf{match}\ e_0\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\} : \widetilde{\mathsf{equate}}_{n\Delta}(G_1, \ldots, G_n)}\ (\text{G-Match})$$

By induction hypothesis we have $\Delta; \Xi; \Gamma \vdash e_0' : G_0'$ with $\Delta \vdash G_0 \sqsubseteq G_0'$ and $\forall i.1 \leq i \leq n, \Delta; \Xi; \Gamma, x_{i1} : G_{i1},\ \ldots,\ x_{im_i} : G_{im_i} \vdash e_i' : G_i'$ with $\Delta \vdash G_i \sqsubseteq G_i'$.

Then $\Delta \vdash G_0' \sim ?_{\mathsf{D}}$ by lemma B.9, $\widetilde{\mathsf{valid}}_{\Delta,\Xi}(\{\overline{p}\}, G_0')$ by lemma B.15 and $\Delta \vdash \widetilde{\mathsf{equate}}_{n\Delta}(G_1, \ldots, G_n) \sqsubseteq \widetilde{\mathsf{equate}}_{n\Delta}(G_1', \ldots, G_n')$ by lemma B.14.

Therefore $\Delta; \Xi; \Gamma \vdash \mathsf{match}\ e_0'\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e'}\} : \widetilde{\mathsf{equate}}_{n\Delta}(G_1', \ldots, G_n')$ and the result holds.

□

Theorem 4.6 (Static Gradual Guarantee). *If* $\Delta; \Xi; \cdot \vdash e : G$ *and* $\Delta \vdash e \sqsubseteq e'$ *then* $\Delta; \Xi; \cdot \vdash e' : G$ *for some* G' *such that* $\Delta \vdash G \sqsubseteq G'$.

Proof. Direct corollary of theorem B.18 □

Appendix C $\lambda^{\varepsilon}_{D?}$ DYNAMIC SEMANTICS

In this section we present the complete definitions for $\lambda^{\varepsilon}_{D?}$ together with the proofs for $\lambda_{D?}$'s dynamic semantics.

C.1 Definitions

We present the new syntax introduced by $\lambda^{\varepsilon}_{D?}$ together with its typing rules (Figure 24) and its dynamic semantics (Figure 25).

$$
\begin{array}{llll}
\textit{Evidence} & \varepsilon & \in & GType \\
\textit{Expressions} & e & ::= & x \mid k \mid e\ e \mid \lambda x{:}G.\ e \mid \varepsilon\, e :: G \mid c\ \left\{\bar{l} = \bar{e}\right\} \mid \mathsf{match}\ e\ \mathsf{with}\ \{\bar{p} \mapsto \bar{e}\} \\
\textit{Values} & v & ::= & \varepsilon\, u :: G \\
\textit{Raw Values} & u & ::= & k \mid \lambda x{:}G.\ e \mid c\ \left\{\bar{l} = \bar{v}\right\} \\
\textit{Errors} & \mathbf{err} & ::= & \mathbf{error_M} \mid \mathbf{error_A} \mid \mathbf{error_T}
\end{array}
$$

Typing rules $\boxed{\Delta; \Xi; \Gamma \vdash e : G}$

$$\frac{\Delta; \Xi \vdash \Gamma \qquad G \doteq \Gamma(x)}{\Delta; \Xi; \Gamma \vdash x : G}\ \varepsilon\text{G-Var} \qquad \frac{}{\Delta; \Xi; \Gamma \vdash k : \mathsf{ty}(k)}\ \varepsilon\text{G-Const}$$

$$\frac{\Delta; \Xi; \Gamma \vdash e_1 : G \qquad \Delta; \Xi; \Gamma \vdash e_2 : \widetilde{\mathsf{dom}_\Delta}(G)}{\Delta; \Xi; \Gamma \vdash e_1\ e_2 : \widetilde{\mathsf{cod}_\Delta}(G)}\ \varepsilon\text{G-App} \qquad \frac{\Delta; \Xi; \Gamma, x : G_x \vdash e : G}{\Delta; \Xi; \Gamma \vdash \lambda x{:}G_x.\ e : G_x \to G}\ \varepsilon\text{G-Lam}$$

$$\frac{\Delta; \Xi; \Gamma \vdash e : G \qquad \varepsilon \Vdash \Delta \vdash G \sim G'}{\Delta; \Xi; \Gamma \vdash \varepsilon\, e :: G' : G'}\ \varepsilon\text{G-Ascribe} \qquad \frac{\begin{array}{c}\widetilde{\Delta; \Xi; \Gamma \vdash \bar{e} : \widetilde{\mathsf{lty}_\Xi}(\bar{l}, c)}\\ \mathsf{satisfylabels}_{n\Xi}(c, l_1 \times \cdots \times l_n)\end{array}}{\Delta; \Xi; \Gamma \vdash c\ \left\{\bar{l}^n = \bar{e}^n\right\} : \widetilde{\mathsf{cty}_\Delta}(c)}\ \varepsilon\text{G-Ctor}$$

$$\frac{\Delta; \Xi; \Gamma \vdash e : G \qquad \Delta \vdash G \sqsubseteq ?_D}{\Delta; \Xi; \Gamma \vdash e.l : \widetilde{\mathsf{fty}_{\Delta,\Xi}}(l, G)}\ \varepsilon\text{G-Access} \qquad \frac{\begin{array}{c}\Delta; \Xi; \Gamma \vdash e : G' \qquad \Delta \vdash G' \sqsubseteq ?_D \qquad \widetilde{\mathsf{valid}_{\Delta,\Xi}}(\{\bar{p}\}, G')\\ \forall i \in [1, n]\ (x_{i1} : G_{i1}) \times \cdots \times (x_{im_i} : G_{im_i}) \doteq \mathsf{parg}_{m_i\Xi}(p_i)\\ \Delta; \Xi; \Gamma, x_{i1} : G_{i1},\ \ldots,\ x_{im_i} : G_{im_i} \vdash e_i : G\end{array}}{\Delta; \Xi; \Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ \{\bar{p} \mapsto \bar{e}\} : G}\ \varepsilon\text{G-Match}$$

Fig. 24. $\lambda^{\varepsilon}_{D?}$: Syntax and typing rules

Reductions $\boxed{e \longrightarrow e \text{ or } \mathbf{err}}$

$$
\begin{array}{llcl}
(\varepsilon\text{R-Beta}) & (\varepsilon_1\ (\lambda x{:}G_{11}.\ e) :: G_1 \to G_2)\ (\varepsilon_2\ u :: G_1) & \longrightarrow & \begin{cases} \mathsf{cod}(\varepsilon_1)(e\,[{}^{(\varepsilon_2 \circ \mathsf{dom}(\varepsilon_1))\ u\ ::\ G_{11}}/_x]) :: G_2 \\ \mathbf{error_T} \quad \text{if } \varepsilon_2 \circ \mathsf{dom}(\varepsilon_1) \text{ is not defined} \end{cases} \\
(\varepsilon\text{R-Delta}) & (\varepsilon_1\ k :: G_1 \to G_2)\ (\varepsilon_2\ u :: G_1) & \longrightarrow & \mathsf{cod}(\varepsilon_1)\ \delta(k, u) :: G_2 \\
(\varepsilon\text{R-AscErase}) & \varepsilon_2\ (\varepsilon_1\ u :: G_1) :: G_2 & \longrightarrow & \begin{cases} (\varepsilon_1 \circ \varepsilon_2)\ u :: G_2 \\ \mathbf{error_T} \quad \text{if } \varepsilon_1 \circ \varepsilon_2 \text{ is not defined} \end{cases} \\
(\varepsilon\text{R-Match}) & \mathsf{match}\ \varepsilon\ c\ \left\{\overline{l} = \overline{v}\right\} :: G\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\} & \longrightarrow & \begin{cases} e_k\,[\overline{v}/\overline{x_k}] & k \text{ smallest } \mathbb{N} \text{ s.t. } c = c_k \\ & \text{where } c_k\ \overline{x_k} \doteq p_k \\ \mathbf{error_M} & \text{there is no } k \text{ s.t. } c = c_k \end{cases} \\
(\varepsilon\text{R-Access}) & (\varepsilon\ c\left\{\overline{l} = \overline{v}\right\} :: G).l & \longrightarrow & \begin{cases} (\varepsilon_k \circ \varepsilon_{G'})\ u_k :: G' & l_k = l \text{ with } \varepsilon_k\ u_k :: G_k \doteq v_k \\ & \text{and } G' \doteq \widetilde{\mathsf{fty}}_{\Delta,\Xi}(l_k, G) \\ \mathbf{error_T} & \text{if } \varepsilon_k \circ \varepsilon_{G'} \text{ is not defined} \\ \mathbf{error_A} & \text{otherwise} \end{cases}
\end{array}
$$

Frames $\boxed{e \longmapsto e \text{ or } \mathbf{err}}$

$$
E \quad ::= \quad \varepsilon\ \square :: G \ \mid\ \square\ e \ \mid\ v\ \square \ \mid\ \square.l \ \mid\ c\ \left\{\overline{l} = \overline{v}, l = \square, \overline{l} = \overline{e}\right\} \ \mid\ \mathsf{match}\ \square\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\}
$$

$$
\frac{e \longrightarrow e'}{e \longmapsto e'}\ \varepsilon\text{R}\longrightarrow \qquad \frac{e \longmapsto e'}{E[e] \longmapsto E[e']}\ \varepsilon\text{RE}
$$

$$
\frac{e \longrightarrow \mathbf{err}}{e \longmapsto \mathbf{err}}\ \varepsilon\text{RERR} \qquad \frac{e \longmapsto \mathbf{err}}{E[e] \longmapsto \mathbf{err}}\ \varepsilon\text{RERR}
$$

Fig. 25. $\lambda^{\varepsilon}_{\mathrm{D?}}$: Dynamic semantics

C.2 Type Safety

Lemma C.1 (Canonical Forms). *Consider a value $\Delta; \Xi; \cdot \vdash v : G$. Then $v = \varepsilon\ u :: G$, with $\Delta; \Xi; \cdot \vdash u : G'$ and $\varepsilon \Vdash \Delta \vdash G' \sim G$. Furthermore:*

(1) If $G = B$, then $v = \varepsilon_B\ k :: B$, with $\Delta; \Xi; \cdot \vdash k : B$ and $\varepsilon_B \Vdash \Delta \vdash B \sim B$.

(2) If $G = G_1 \to G_2$, then $v = \varepsilon\ (\lambda x{:}G_1'.\ e) :: G_1 \to G_2$ with $\Delta; \Xi; x : G_1' \vdash e : G_2'$ and $\varepsilon \Vdash \Delta \vdash G_1' \to G_2' \sim G_1 \to G_2$, or $\varepsilon\ k :: G_1 \to G_2$ with $\Delta; \Xi; \cdot \vdash k : G_1' \to G_2'$ and $\varepsilon \Vdash \Delta \vdash G_1' \to G_2' \sim G_1 \to G_2$.

(3) If $G \in \{D, ?_{\mathrm{D}}, ?_{\mathrm{O}}\}$, then $v = \varepsilon\ c\ \left\{\overline{l} = \overline{v}\right\} :: G$, with $\Delta; \Xi; \cdot \vdash c\ \left\{\overline{l} = \overline{v}\right\} : G'$ and $\varepsilon \Vdash \Delta \vdash G' \sim G$.

Proof. By direct inspection of the formation rules of evidence terms. □

Lemma C.2 (Substitution). *If $\Delta; \Xi; \Gamma, x : G_x \vdash e : G$, and $\Delta; \Xi; \cdot \vdash v : G_x$, then $\Delta; \Xi; \Gamma \vdash e\,[v/x] : G$.*

Proof. By induction on the derivation of $\Delta; \Xi; \Gamma, x : G_x \vdash e : G$. Only interesting cases are shown.

Case (εG-Ctor)

Then

$$\frac{\Delta;\Xi;\Gamma,x:G_x\vdash\overline{e}:\widetilde{\mathsf{lty}_\Xi}(\overline{l},c)\quad \mathsf{satisfylabels}_{n\Xi}(c,l_1\times\cdots\times l_n)}{\Delta;\Xi;\Gamma,x:G_x\vdash c\ \left\{\overline{l}=\overline{e}\right\}:\widetilde{\mathsf{cty}_\Delta}(c)}\ (\varepsilon\text{G-Ctor})$$

By induction hypothesis $\Delta;\Xi;\Gamma\vdash\overline{e}\,[v/x]:\widetilde{\mathsf{lty}_\Xi}(\overline{l},c)$.

Then $\Delta;\Xi;\Gamma\vdash c\ \left\{\overline{l}=\overline{e}\,[v/x]\right\}:G$.

Therefore $\Delta;\Xi;\Gamma\vdash c\ \left\{\overline{l}=\overline{e}\right\}[v/x]:G$ and the result holds.

Case (εG-Access)

Then

$$\frac{\Delta;\Xi;\Gamma,x:G_x\vdash e_0:G_0\quad \Delta\vdash G_0\sqsubseteq ?_{\mathsf{D}}}{\Delta;\Xi;\Gamma,x:G_x\vdash e_0.l:\widetilde{\mathsf{fty}_{\Delta,\Xi}}(l,G_0)}\ (\varepsilon\text{G-Access})$$

By induction hypothesis $\Delta;\Xi;\Gamma\vdash e_0\,[v/x]:G_0$.

Then $\Delta;\Xi;\Gamma\vdash e_0\,[v/x].l:\widetilde{\mathsf{fty}_{\Delta,\Xi}}(l,G_0)$.

Therefore $\Delta;\Xi;\Gamma\vdash(e_0.l)\,[v/x]:\widetilde{\mathsf{fty}_{\Delta,\Xi}}(l,G_0)$ and the result holds.

Case (εG-Match)

Then

$$\frac{\Delta;\Xi;\Gamma,x:G_x\vdash e_0:G_0\quad \begin{array}{c}\Delta\vdash G_0\sqsubseteq ?_{\mathsf{D}}\\ \widetilde{\mathsf{valid}_{\Delta,\Xi}}(\{\overline{p}\},G_0)\end{array}\quad \begin{array}{c}\forall i.1\leq i\leq n\\ (x_{i1}:G_{i1})\times\cdots\times(x_{im_i}:G_{im_i})\doteq\widetilde{\mathsf{parg}_{m_i\Xi}}(p_i)\\ \Delta;\Xi;\Gamma,x:G_x,x_{i1}:G_{i1},\ldots,x_{im_i}:G_{im_i}\vdash e_i:G\end{array}}{\Delta;\Xi;\Gamma\vdash\mathsf{match}\ e_0\ \mathsf{with}\ \{\overline{p}\mapsto\overline{e}\}:G}\ (\varepsilon\text{G-Match})$$

By induction hypothesis $\Delta;\Xi;\Gamma\vdash e_0\,[v/x]:G_0$ and

$\Delta;\Xi;\Gamma,x_{i1}:G_{i1},\ldots,x_{im_i}:G_{im_i}\vdash e_i\,[v/x]:G$.

Then $\Delta;\Xi;\Gamma\vdash\mathsf{match}\ e_0\,[v/x]\ \mathsf{with}\ \{\overline{p}\mapsto\overline{e}\,[v/x]\}:G$.

Therefore $\Delta;\Xi;\Gamma\vdash(\mathsf{match}\ e_0\ \mathsf{with}\ \{\overline{p}\mapsto\overline{e}\})\,[v/x]:G$ and the result holds.

□

Lemma C.3. *If* $\varepsilon\doteq\mathsf{initial}_\Delta(G,G')$ *then* $\varepsilon\Vdash\Delta\vdash G\sim G'$.

Proof. Expanding the definition of $\mathsf{initial}_\Delta$

$$\begin{aligned}\varepsilon &= \mathsf{initial}_\Delta(G,G')\\ &= \alpha_\Delta(\gamma_\Delta(G)\cap\gamma_\Delta(G'))\\ &= \alpha_\Delta(\{T\ \mid\ T\in\gamma_\Delta(G),\ T'\in\gamma_\Delta(G'),\ T=T'\})\end{aligned}$$

Because precision is reflexive, we have

$$\Delta \vdash \varepsilon \sqsubseteq \alpha_\Delta(\{T \mid T \in \gamma_\Delta(G),\ T' \in \gamma_\Delta(G'),\ T = T'\})$$

which is the definition of type consistency.

Therefore $\varepsilon \Vdash \Delta \vdash G \sim G'$. □

Lemma C.4. *If* $\Delta; \Xi; \Gamma \vdash e : G$ *and* $e' \doteq \mathsf{ascribe}_\Delta(e, G, G')$ *then* $\Delta; \Xi; \Gamma \vdash e' : G'$.

Proof. By the definition of $\mathsf{ascribe}_\Delta$. □

Lemma C.5. *If* $\widetilde{\mathsf{valid}}_{\Delta,\Xi}(P, G)$*, then* $\widetilde{\mathsf{valid}}_{\Delta,\Xi}(P, \mathsf{todata}_\Delta(G))$.

Proof. By cases on G.

Case ($G \in \{D, ?_O, ?_D\}$)
Then $G = \mathsf{todata}_\Delta(G)$ and the result holds immediately.

Case ($G = ?$) Then $?_D = \mathsf{todata}_\Delta(?)$. Since $\mathsf{valid}_{\Delta,\Xi}$ is only defined for datatypes D, then $\widetilde{\mathsf{valid}}_{\Delta,\Xi}(P, ?) \implies \widetilde{\mathsf{valid}}_{\Delta,\Xi}(P, ?_D)$ and the result holds.

Case (G) Since $\widetilde{\mathsf{valid}}_{\Delta,\Xi}(P, G)$ does not hold the result holds trivially.

□

Lemma C.6. *If* $G' \doteq \widetilde{\mathsf{fty}}_{\Delta,\Xi}(l, G)$, $G' = \widetilde{\mathsf{fty}}_{\Delta,\Xi}(l, \mathsf{todata}_\Delta(G))$.

Proof. By cases on G.

Case $G \in \{D, ?_O, ?_D\}$
By definition, $G = \mathsf{todata}_\Delta(G)$ and the result holds immediately.

Case $G = ?$ We know that $?_D = \mathsf{todata}_\Delta(?)$ and $\widetilde{\mathsf{fty}}_{\Delta,\Xi}(l, ?_D) = \widetilde{\mathsf{fty}}_{\Delta,\Xi}(l, ?)$ by definition, and the result holds.

Case otherwise
Trivial, since $\widetilde{\mathsf{fty}}_{\Delta,\Xi}(l, G)$ is not defined.

□

Lemma C.7. *If* $\Delta \vdash G \sim ?_D$ *then* $\Delta \vdash \mathsf{todata}_\Delta(G) \sqsubseteq ?_D$.

Proof. By cases on G.

Case ($G = D$)
By definition of todata_Δ, $G = \mathsf{todata}_\Delta(G)$. Then $\Delta \vdash G \sqsubseteq ?_D$, and the result holds.

Case ($G = ?_O$)
By definition of todata_Δ, $G = \mathsf{todata}_\Delta(G)$. Then $\Delta \vdash G \sqsubseteq ?_O$, and the result holds.

Case ($G \in \{?_{\mathrm{D}}, ?\}$)

By definition of todata_Δ, $?_{\mathrm{D}} = \mathrm{todata}_\Delta(G)$. Then $\Delta \vdash ?_{\mathrm{D}} \sqsubseteq ?_{\mathrm{D}}$, and the result holds.

Case (G) The judgment $\Delta \vdash G \sim ?_{\mathrm{D}}$ does not hold. Therefore the result holds trivially.

□

Lemma C.8. *If $\varepsilon \Vdash \Delta \vdash G_1 \sim G_2$ and $\varepsilon \circ \varepsilon_G$ is defined, then $\varepsilon \circ \varepsilon_G \Vdash \Delta \vdash G_1 \sim G$.*

Proof. Unfolding the definitions

$$\begin{array}{rcl} \varepsilon \Vdash \Delta \vdash G_1 \sim G_2 & \Leftrightarrow & \Delta \vdash \varepsilon \sqsubseteq \alpha_\Delta(\{T_1 \mid T_1 \in \gamma_\Delta(G_1), T_2 \in \gamma_\Delta(G_2), T_1 = T_2\}) \\ & \Leftrightarrow & \Delta \vdash \varepsilon \sqsubseteq \alpha_\Delta(\gamma_\Delta(G_1) \cap \gamma_\Delta(G_2)) \end{array}$$

$$\varepsilon \circ \varepsilon_G = \alpha_\Delta(\gamma_\Delta(\varepsilon) \cap \gamma_\Delta(\varepsilon_G))$$

Then

$$\begin{array}{rcl} \Delta \vdash \varepsilon & \sqsubseteq & \alpha_\Delta(\gamma_\Delta(G_1) \cap \gamma_\Delta(G_2)) \\ & \sqsubseteq & \alpha_\Delta(\gamma_\Delta(G_1)) \\ & \sqsubseteq & G_1 \end{array}$$

Then

$$\begin{array}{rcl} \varepsilon & \sqsubseteq & G_1 \\ \gamma_\Delta(\varepsilon) & \subseteq & \gamma_\Delta(G_1) \\ \gamma_\Delta(\varepsilon) \cap \gamma_\Delta(\varepsilon_G) & \subseteq & \gamma_\Delta(G_1) \cap \gamma_\Delta(\varepsilon_G) \\ \alpha_\Delta(\gamma_\Delta(\varepsilon) \cap \gamma_\Delta(\varepsilon_G)) & \sqsubseteq & \alpha_\Delta(\gamma_\Delta(G_1) \cap \gamma_\Delta(\varepsilon_G)) \\ \varepsilon \circ \varepsilon_G & \sqsubseteq & \alpha_\Delta(\{T_1 \mid T_1 \in \gamma_\Delta(G_1), T \in \gamma_\Delta(G), T_1 = T\}) \end{array}$$

Therefore $\varepsilon \circ \varepsilon_G \Vdash \Delta \vdash G_1 \sim G$ and the result holds. □

Lemma C.9 (Preservation for $\longrightarrow$). *If $\Delta; \Xi; \cdot \vdash e : G$ and $e \longrightarrow e'$, then $\Delta; \Xi; \cdot \vdash e' : G$.*

Proof. By induction on the derivation of $e \longrightarrow e'$.

Case (εR-AscErase)

Then $e = \varepsilon_2\ (\varepsilon_1\ u :: G_1) :: G$. Then

$$\dfrac{\dfrac{\Delta; \Xi; \cdot \vdash u : G_u \quad \varepsilon_1 \Vdash \Delta \vdash G_u \sim G_1}{\Delta; \Xi; \cdot \vdash \varepsilon_1\ u :: G_1 : G_1}\ (\varepsilon\text{G-Ascribe}) \quad \varepsilon_2 \Vdash \Delta \vdash G_1 \sim G}{\Delta; \Xi; \cdot \vdash \varepsilon_2\ (\varepsilon_1\ u :: G_1) :: G : G}\ (\varepsilon\text{G-Ascribe})$$

and

$$\dfrac{(\varepsilon_1 \circ \varepsilon_2) \text{ is defined}}{\varepsilon_2\ (\varepsilon_1\ u :: G_1) :: G \longrightarrow (\varepsilon_1 \circ \varepsilon_2)\ u :: G}\ (\varepsilon\text{R-AscErase})$$

where $(\varepsilon_1 \circ \varepsilon_2) \Vdash \Delta \vdash G_u \sim G$. Then

$$\dfrac{\Delta; \Xi; \cdot \vdash u : G_u \quad (\varepsilon_1 \circ \varepsilon_2) \Vdash \Delta \vdash G_u \sim G}{\Delta; \Xi; \cdot \vdash (\varepsilon_1 \circ \varepsilon_2)\ u :: G : G}\ (\varepsilon\text{G-Ascribe})$$

and the result holds.

Case (εR-Access)

Then $e = (\varepsilon\ c\ \left\{\overline{l} = \overline{\varepsilon}\ \overline{u} :: \overline{G}\right\} :: G').l$. Then

$$\dfrac{\dfrac{\begin{array}{c}\Delta;\Xi;\cdot \vdash \overline{v} : \widetilde{\mathsf{lty}_{\Xi}}(\overline{l}, c) \\ \widetilde{\mathsf{satisfylabels}}_{n\Xi}(c, l_1 \times \cdots \times l_n)\end{array}}{\Delta;\Xi;\cdot \vdash c\ \left\{\overline{l} = \overline{v}\right\} : \widetilde{\mathsf{cty}_{\Delta}}(c)}\ (\varepsilon\text{G-Ctor}) \quad \varepsilon \Vdash \Delta \vdash \widetilde{\mathsf{cty}_{\Delta}}(c) \sim G'}{\Delta;\Xi;\cdot \vdash \varepsilon\ c\ \left\{\overline{l} = \overline{v}\right\} :: G' : G'}\ (\varepsilon\text{G-Ascribe}) \quad \Delta \vdash G' \sqsubseteq ?_{\mathrm{D}}}{\Delta;\Xi;\cdot \vdash (\varepsilon\ c\ \left\{\overline{l} = \overline{v}\right\} :: G').l : \widetilde{\mathsf{fty}_{\Delta,\Xi}}(l, G')}\ (\varepsilon\text{G-Access})$$

and

$$\dfrac{l_k = l \quad v_k \doteq \varepsilon_k\ u_k :: G_k \quad G \doteq \widetilde{\mathsf{fty}_{\Delta,\Xi}}(l, G') \quad (\varepsilon_k \circ \varepsilon_G) \text{ is defined}}{(\varepsilon\ c\ \left\{\overline{l} = \overline{v}\right\} :: G').l \longrightarrow (\varepsilon_k \circ \varepsilon_G)\ u_k :: G}\ (\varepsilon\text{R-Access})$$

With $G_k \doteq \widetilde{\mathsf{lty}_{\Xi}}(l_k, c)$. Then

$$\dfrac{\Delta;\Xi;\cdot \vdash u_k : G'_k \quad \varepsilon_k \Vdash \Delta \vdash G'_k \sim G_k}{\Delta;\Xi;\cdot \vdash \varepsilon_k\ u_k :: G_k : G_k}\ (\varepsilon\text{G-Ascribe})$$

Then $(\varepsilon_k \circ \varepsilon_G) \Vdash \Delta \vdash G'_k \sim G$ since transitivity is defined and lemma C.8.

Finally

$$\dfrac{\Delta;\Xi;\cdot \vdash u_k : G'_k \quad \varepsilon_k \circ \varepsilon_G \Vdash \Delta \vdash G'_k \sim G}{\Delta;\Xi;\cdot \vdash (\varepsilon_k \circ \varepsilon_G)\ u_k :: G : G}\ (\varepsilon\text{G-Ascribe})$$

and the result holds.

Case (εR-Match)

Then $e = \mathsf{match}\ \varepsilon\ c\ \left\{\overline{l} = \overline{v}\right\} :: G'\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\}$. Then

$$\dfrac{\begin{array}{c}\Delta \vdash G' \sqsubseteq ?_{\mathrm{D}} \\ \widetilde{\mathsf{valid}}_{\Delta,\Xi}(\{\overline{p}\}, G') \\ (\forall i.1 \leq i \leq n) \\ (x_{i1} : G_{i1}) \times \cdots \times (x_{im_i} : G_{im_i}) \doteq \mathsf{parg}_{m_i\Xi}(p_i) \\ \Delta;\Xi; x_{i1} : G_{i1},\ \ldots,\ x_{im_i} : G_{im_i} \vdash e_i : G\end{array} \quad \dfrac{\varepsilon \Vdash \Delta \vdash G'' \sim G' \quad \dfrac{\begin{array}{c}G'' \doteq \widetilde{\mathsf{cty}_{\Delta}}(c) \quad \Delta;\Xi;\cdot \vdash \overline{v} : \widetilde{\mathsf{lty}_{\Xi}}(\overline{l}, c) \\ \mathsf{satisfylabels}_{n\Xi}(c, l_1 \times \cdots \times l_n)\end{array}}{\Delta;\Xi;\cdot \vdash c\ \left\{\overline{l} = \overline{v}\right\} : G''}\ (\varepsilon\text{G-Ctor})}{\Delta;\Xi;\cdot \vdash \varepsilon\ c\ \left\{\overline{l} = \overline{v}\right\} :: G' : G'}\ (\varepsilon\text{G-Ascribe})}{\Delta;\Xi;\cdot \vdash \mathsf{match}\ \varepsilon\ c\ \left\{\overline{l} = \overline{v}\right\} :: G'\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\} : G}\ (\varepsilon\text{G-Match})$$

and
$$\frac{k \text{ is the smallest } \mathbb{N} \text{ s.t } c = c_k \text{ with } c_k \overline{x_k} \doteq p_k}{\text{match } \varepsilon\ c\ \left\{\overline{l} = \overline{v}\right\} :: G' \text{ with } \{\overline{p} \mapsto \overline{e}\} \longrightarrow e_k\ [\overline{v}/\overline{x_k}]}\ (\varepsilon\text{R-Match})$$

By the definitions of $\widetilde{\mathsf{parg}}_{n\Xi}$ and $\widetilde{\mathsf{lty}}_{\Xi}$ we have

$$\widetilde{\mathsf{parg}}_{n\Xi}(c\ \overline{x}) = (x_1 : \widetilde{\mathsf{lty}}_{\Xi}(l_1, c)) \times \cdots \times (x_n : \widetilde{\mathsf{lty}}_{\Xi}(l_n, c))$$

Then we have

$$\Delta; \Xi; \cdot \vdash e_k\ [\overline{v}/\overline{x_k}] : G$$

by lemma C.2 and the result holds.

□

Lemma C.10 (Preservation). *If* $\Delta; \Xi; \cdot \vdash e : G$ *and* $e \longmapsto e'$ *, then* $\Delta; \Xi; \cdot \vdash e' : G$.

Proof. By induction on the derivation of $e \longmapsto e'$.

Case (R $\longrightarrow$) Then
$$\frac{e \longrightarrow e'}{e \longmapsto e'}\ (\text{R} \longrightarrow)$$

and the result holds immediately by lemma C.9.

Case (RE) Then $e = E[e_0]$ and we have
$$\frac{e_0 \longmapsto e_0'}{E[e_0] \longmapsto E[e_0']}\ (\text{RE})$$

Since $\Delta; \Xi; \cdot \vdash E[e_0] : G$ we know that $\Delta; \Xi; \cdot \vdash e_0 : G_0$ with $E : G_0 \to G$.

Then, by induction hypothesis we have $\Delta; \Xi; \cdot \vdash e_0' : G_0$.

Therefore $\Delta; \Xi; \cdot \vdash E[e_0'] : G$ and the result holds.

□

Lemma C.11 (Progress). *If* $\Delta; \Xi; \cdot \vdash e : G$ *, then either*

- e *is a value*
- $\exists e'$ *s.t.* $e \longmapsto e'$
- $e \longmapsto$ **err**

Proof. By induction on the derivation of $\Delta; \Xi; \cdot \vdash e : G$.

Case (εG-Ascribe)

Then $e = \varepsilon\ e_0 :: G$ and we have
$$\frac{\Delta; \Xi; \cdot \vdash e_0 : G_0 \quad \varepsilon \Vdash \Delta \vdash G_0 \sim G}{\Delta; \Xi; \cdot \vdash \varepsilon\ e_0 :: G : G}\ (\varepsilon\text{G-Ascribe})$$

By induction hypothesis, either e_0 is a value, $e_0 \longmapsto e_0'$ or $e_0 \longmapsto$ **err**.

- If e_0 is a value, then $e_0 \doteq \varepsilon_0\ u :: G_0$. Then $e \longmapsto r$ by εR-AscErase.

- If $e_0 \longmapsto e_0'$, then $\varepsilon\ e_0 :: G \longmapsto \varepsilon\ e_0' :: G$ by εRE.
- If $e_0 \longmapsto \mathbf{err}$, then $e \longmapsto \mathbf{err}$ by εRErr.

Case (εG-Ctor)

Then $e = c\ \left\{\overline{l} = \overline{e}\right\}$. If e is a value, then the result holds immediately. If e is not a value, then at least one of it's subexpressions is not a value either. Let k be the smaller index of the subexpressions which are not values.

Then

$$\frac{\mathsf{satisfylabels}_{n\Xi}(c, l_1 \times \cdots \times l_n) \quad \Delta; \Xi; \cdot \vdash \overline{e} : \widetilde{\mathsf{lty}_\Xi}(\overline{l}, c)}{\Delta; \Xi; \cdot \vdash c\ \left\{\overline{l} = \overline{e}\right\} : \widetilde{\mathsf{cty}_\Delta}(c)}\ (\varepsilon\text{G-Ctor})$$

in particular $\Delta; \Xi; \cdot \vdash e_k : \widetilde{\mathsf{lty}_\Xi}(l_k, c)$.

By induction hypothesis, we have either $e_k \longmapsto e_k'$ or $e_k \longmapsto \mathbf{err}$, since e_k is not a value by defiition.

- If $e_k \longmapsto e_k'$, then $e \longmapsto e'$ with $e' = c\ \left\{\overline{l} = \overline{v}, l_k = e_k', \overline{l} = \overline{e}\right\}$ by εRE.
- If $e_k \longmapsto \mathbf{err}$ then $e \longmapsto \mathbf{err}$ by εRErr.

And the result holds.

Case (εG-Access)

Then $e = e_0.l$ and we have

$$\frac{\Delta; \Xi; \cdot \vdash e_0 : G_0 \quad \Delta \vdash G_0 \sqsubseteq ?_\mathrm{D}}{\Delta; \Xi; \cdot \vdash e_0.l : \widetilde{\mathsf{fty}_{\Delta,\Xi}}(l, G_0)}\ (\varepsilon\text{G-Access})$$

By induction hypothesis, we know that either e_0 is a value, $e_0 \longmapsto e_0'$ or $e_0 \longmapsto \mathbf{err}$.

- If e_0 is value, then by Canonical Forms (Lemma C.1), $e_0 \doteq \varepsilon\ c\ \left\{\overline{l} = \overline{v}\right\} :: G_0$ since $\Delta \vdash G_0 \sqsubseteq ?_\mathrm{D}$.

  Then, $e \longmapsto r$, by εR-Access.
- If $e_0 \longmapsto e_0'$, then $e_0.l \longmapsto e_0'.l$ by εRE.
- If $e_0 \longmapsto \mathbf{err}$, then $e \longmapsto \mathbf{err}$ by εRErr.

And the result holds.

Case (εG-Match)

Then $e = \mathsf{match}\ e_0\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\}$ and we have

$$\frac{\Delta; \Xi; \cdot \vdash e_0 : G_0 \quad \Delta \vdash G_0 \sqsubseteq ?_\mathrm{D} \quad \widetilde{\mathsf{valid}_{\Delta,\Xi}}(\left\{\overline{p}\right\}, G_0) \quad \begin{array}{c} \forall i. 1 \leq i \leq n \\ (x_{i1} : G_{i1}) \times \cdots \times (x_{im_i} : G_{im_i}) \doteq \widetilde{\mathsf{parg}_{m_i\Xi}}(p_i) \\ \Delta; \Xi; \Gamma, x_{i1} : G_{i1}, \ldots, x_{im_i} : G_{im_i} \vdash e_i : G \end{array}}{\Delta; \Xi; \cdot \vdash \mathsf{match}\ e_0\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\} : G}\ (\varepsilon\text{G-Match})$$

By induction hypothesis, either e_0 is a value, $e_0 \longmapsto e_0'$ or $e_0 \longmapsto \mathbf{err}$.

- If e_0 is a value, then by Canonical Forms (Lemma C.1), $e_0 \doteq \varepsilon\ c\ \left\{\overline{l} = \overline{v}\right\} :: G_0$, since $\Delta \vdash G_0 \sqsubseteq ?_{\mathrm{D}}$.

  Then $e_0 \longmapsto r$ by εR-Match.
- If $e_0 \longmapsto e_0'$, then match e_0 with $\{\overline{p} \mapsto \overline{e}\} \longmapsto$ match e_0' with $\{\overline{p} \mapsto \overline{e}\}$ by εRE.
- If $e_0 \longmapsto \mathbf{err}$, then $e \longmapsto \mathbf{err}$ by εRErr.

And the result holds.

□

Theorem 4.3 (Type Safety). *If $\Delta; \Xi; \cdot \vdash e : G$, then either $e \Downarrow v$ with $\Delta; \Xi; \cdot \vdash v : G$, or $e \Downarrow \mathbf{err}$.*

Proof. Direct from Progress and Preservation (Lemmas C.10 and C.11) using the fact that the translation function preserves typing (Theorem C.24). □

C.3 Dynamic Gradual Guarantee

$$\frac{\Delta \vdash G_1 \sqsubseteq G_2 \qquad \Delta \vdash e_1 \sqsubseteq e_2 \qquad \Delta \vdash \varepsilon_1 \sqsubseteq \varepsilon_2}{\Delta \vdash \varepsilon_1\ e_1 :: G_1 \sqsubseteq \varepsilon_2\ e_2 :: G_2}\ \text{P-}\varepsilon\text{Asc}$$

Lemma C.12 (Dynamic Guarantee for $\longrightarrow$). *If $\Delta; \Xi; \Gamma \vdash e_1 : G$ and $e_1 \longrightarrow e_2$, then for any e_1' such that $\Delta \vdash e_1 \sqsubseteq e_1'$, we have $e_1' \longrightarrow e_2'$ for some e_2' such that $\Delta \vdash e_2 \sqsubseteq e_2'$.*

Proof. Let e_1' such that $\Delta \vdash e_1 \sqsubseteq e_1'$. By induction on $e_1 \longrightarrow e_2$.

Case (εR-AscErase)

We know that $e_1 = \varepsilon_2\ (\varepsilon_1\ u :: G_1) :: G$. Then

$$\dfrac{\dfrac{\Delta \vdash u \sqsubseteq u' \quad \Delta \vdash \varepsilon_1 \sqsubseteq \varepsilon_3 \quad \Delta \vdash G_1 \sqsubseteq G_1'}{\Delta \vdash \varepsilon_1\ u :: G_1 \sqsubseteq \varepsilon_3\ u' :: G_1'}\ (\text{P-}\varepsilon\text{Asc}) \quad \Delta \vdash \varepsilon_2 \sqsubseteq \varepsilon_4 \quad \Delta \vdash G \sqsubseteq G'}{\Delta \vdash \varepsilon_2\ (\varepsilon_1\ u :: G_1) :: G \sqsubseteq \varepsilon_4\ (\varepsilon_3\ u' :: G_1') :: G'}\ (\text{P-}\varepsilon\text{Asc})$$

By εR-AscErase

$$\varepsilon_2\ (\varepsilon_1\ u :: G_1) :: G \longrightarrow (\varepsilon_1 \circ \varepsilon_2)\ u :: G$$

with $(\varepsilon_1 \circ \varepsilon_2)$ defined.

Since consistent transitivity is monotonic we have $\Delta \vdash (\varepsilon_1 \circ \varepsilon_2) \sqsubseteq (\varepsilon_3 \circ \varepsilon_4)$.

Then

$$\varepsilon_4\ (\varepsilon_3\ u' :: G_1') :: G' \longrightarrow (\varepsilon_3 \circ \varepsilon_4)\ u' :: G'$$

Therefore $\Delta \vdash (\varepsilon_1 \circ \varepsilon_2)\, u :: G \sqsubseteq (\varepsilon_3 \circ \varepsilon_4)\, u' :: G'$ and the result holds.

Case (εR-Match)

We know that $e_1 = \mathsf{match}\ \varepsilon_1\ c\ \left\{\overline{l = v}\right\} :: G_0\ \mathsf{with}\ \{\overline{p \mapsto e}\}$. Then

$$\dfrac{\dfrac{\dfrac{\Delta \vdash \overline{v} \sqsubseteq \overline{v'}}{\Delta \vdash c\ \left\{\overline{l = v}\right\} \sqsubseteq c\ \left\{\overline{l = v'}\right\}}\ (\text{P-Ctor}) \quad \Delta \vdash \varepsilon_1 \sqsubseteq \varepsilon_2 \quad \Delta \vdash G_0 \sqsubseteq G_0'}{\Delta \vdash \varepsilon_1\ c\ \left\{\overline{l = v}\right\} :: G_0 \sqsubseteq \varepsilon_2\ c\ \left\{\overline{l = v'}\right\} :: G_0'}\ (\text{P-}\varepsilon\text{Asc}) \quad \Delta \vdash \overline{e} \sqsubseteq \overline{e'}}{\Delta \vdash \mathsf{match}\ \varepsilon_1\ c\ \left\{\overline{l = v}\right\} :: G_0\ \mathsf{with}\ \{\overline{p \mapsto e}\} \sqsubseteq \mathsf{match}\ \varepsilon_2\ c\ \left\{\overline{l = v'}\right\} :: G_0'\ \mathsf{with}\ \{\overline{p \mapsto e'}\}}\ (\text{P-Match})$$

By εR-Match

$$\mathsf{match}\ \varepsilon_1\ c\ \left\{\overline{l = v}\right\} :: G_0\ \mathsf{with}\ \{\overline{p \mapsto e}\} \longrightarrow e_k\ [\overline{v}/\overline{x_k}]$$

with k the smallest integer such that $c = c_k$ where $c_k\ \overline{x_k} \doteq p_k$.

Then

$$\mathsf{match}\ \varepsilon_2\ c\ \left\{\overline{l = v'}\right\} :: G_0'\ \mathsf{with}\ \{\overline{p \mapsto e'}\} \longrightarrow e_k'\ [\overline{v'}/\overline{x_k}]$$

By lemma B.17, we have $\Delta \vdash e_k\ [\overline{v}/\overline{x_k}] \sqsubseteq e_k'\ [\overline{v'}/\overline{x_k}]$ and the result holds.

Case (εR-Access)

We known that $e_1 = \varepsilon_1\ c\ \left\{\overline{l = v}\right\} :: G_0.l$. Then

$$\dfrac{\dfrac{\dfrac{\Delta \vdash \overline{v} \sqsubseteq \overline{v'}}{\Delta \vdash c\ \left\{\overline{l = v}\right\} \sqsubseteq c\ \left\{\overline{l = v'}\right\}}\ (\text{P-Ctor}) \quad \Delta \vdash \varepsilon_1 \sqsubseteq \varepsilon_2 \quad \Delta \vdash G_0 \sqsubseteq G_0'}{\Delta \vdash \varepsilon_1\ c\ \left\{\overline{l = v}\right\} :: G_0 \sqsubseteq \varepsilon_2\ c\ \left\{\overline{l = v'}\right\} :: G_0'}\ (\text{P-}\varepsilon\text{Asc})}{\Delta \vdash \varepsilon_1\ c\ \left\{\overline{l = v}\right\} :: G_0.l \sqsubseteq \varepsilon_2\ c\ \left\{\overline{l = v'}\right\} :: G_0'.l}\ (\text{P-Access})$$

By εR-Access

$$\varepsilon_1\ c\ \left\{\overline{l = v}\right\} :: G_0.l \longrightarrow (\varepsilon_1 \circ \varepsilon_G)\, u_k :: G$$

with $l = l_k$, $v_k \doteq \varepsilon_k\ u_k :: G_k$, $G \doteq \widetilde{\mathsf{fty}}_{\Delta,\Xi}(l, G_0)$ and $(\varepsilon_1 \circ \varepsilon_G)$ defined.

Let $G' \doteq \widetilde{\mathsf{fty}_{\Delta,\Xi}}(l, G'_0)$. Then $\Delta \vdash G \sqsubseteq G'$ by lemma B.13.

Since consistent transitivity is monotonic, we have $\Delta \vdash \varepsilon_1 \circ \varepsilon_G \sqsubseteq \varepsilon_2 \circ \varepsilon_{G'}$.

Then

$$\varepsilon_2\ c\ \left\{\overline{l} = \overline{v'}\right\} :: G'_0.l \longrightarrow (\varepsilon_2 \circ \varepsilon_{G'})\ u'_k :: G'$$

Finally, $\Delta \vdash (\varepsilon_1 \circ \varepsilon_G)\ u_k :: G \sqsubseteq (\varepsilon_2 \circ \varepsilon_{G'})\ u'_k :: G'$ and the result holds.

□

Lemma C.13 (Dynamic Gradual Guarantee for $\longmapsto$). *If $\Delta; \Xi; \Gamma \vdash e_1 : G$ and $e_1 \longmapsto e_2$, then for any e'_1 such that $\Delta \vdash e_1 \sqsubseteq e'_1$, we have $e'_1 \longmapsto e'_2$ for some e'_2 such that $\Delta \vdash e_2 \sqsubseteq e'_2$.*

Proof. By induction on the derivation of $e_1 \longmapsto e_2$.

Case (R $\longrightarrow$) Then

$$\frac{e_1 \longrightarrow e_2}{e_1 \longmapsto e_2}\ (\text{R} \longrightarrow)$$

and the result holds immediately by lemma C.12.

Case (RE) Then $e_1 = E[e_{11}]$ and we have

$$\frac{e_{11} \longmapsto e_{21}}{E[e_{11}] \longmapsto E[e_{21}]}\ (\text{RE})$$

Since $\Delta; \Xi; \Gamma \vdash E[e_{11}] : G$ we know that $\Delta; \Xi; \Gamma \vdash e_{11} : G_{11}$ with $E : G_{11} \rightarrow G$.

We also know that

$$\frac{\Delta \vdash E \sqsubseteq E' \quad \Delta \vdash e_{11} \sqsubseteq e'_{11}}{\Delta \vdash E[e_{11}] \sqsubseteq E'[e'_{11}]}$$

denoting by $\Delta \vdash E \sqsubseteq E'$ the rest of the derivation tree.

Then, by induction hypothesis we have $e'_{11} \longmapsto e'_{21}$ with $\Delta \vdash e_{21} \sqsubseteq e'_{21}$.

Then

$$\frac{\Delta \vdash E \sqsubseteq E' \quad \Delta \vdash e_{21} \sqsubseteq e'_{21}}{\Delta \vdash E[e_{21}] \sqsubseteq E'[e'_{21}]}$$

Finally $E'[e'_{11}] \longmapsto E'[e'_{21}]$ and the result holds.

□

Theorem 4.7 (Dynamic Gradual Guarantee). *If* $\Delta; \Xi; \cdot \vdash e : G$ *and* $e \Downarrow v$*, then for any* e' *such that* $\Delta \vdash e \sqsubseteq e'$*, we have* $e' \Downarrow v'$ *for some* v' *such that* $\Delta \vdash v \sqsubseteq v'$*.*

Proof. Direct by Dynamic Gradual Guarantee for $\longmapsto$ (Lemma C.13) and Translation preserves typing (Theorem C.24). □

C.4 Static Terms Do Not Fail More

Since $T_1 = T_2$ coincides with $\Delta \vdash T_1 \sim T_2$ for static types T_1 an T_2, we would use the former notation in the proofs of this section.

Lemma C.14. *If* ε_1 *and* ε_2 *are static evidences and* $(\varepsilon_1 \circ \varepsilon_2)$ *is defined, then* $(\varepsilon_1 \circ \varepsilon_2) = \varepsilon_1$*.*

Proof. By the definition of consistent transitivity

$$\begin{array}{rcll} \varepsilon_1 \circ \varepsilon_2 & = & \alpha_\Delta(\gamma_\Delta(\varepsilon_1) \cap \gamma_\Delta(\varepsilon_2)) & \\ & = & \alpha_\Delta(\{\varepsilon_1\} \cap \{\varepsilon_2\}) & \\ & = & \alpha_\Delta(\{\varepsilon_1\}) & \text{Since it's defined} \\ & = & \varepsilon_1 & \end{array}$$

And the result holds. □

Lemma C.15. *If* ε_1 *and* ε_2 *are static evidences such that* $\varepsilon_1 \Vdash \Delta \vdash T_1 \sim T_2$ *and* $\varepsilon_2 \Vdash \Delta \vdash T_2 \sim T_3$ *then* $(\varepsilon_1 \circ \varepsilon_2)$ *is defined and static.*

Proof. By the definition of consistency

$$\begin{array}{rcl} \varepsilon_1 \Vdash \Delta \vdash T_1 \sim T_2 & \Leftrightarrow & \Delta \vdash \varepsilon_1 \sqsubseteq \alpha_\Delta(\gamma_\Delta(T_1) \cap \gamma_\Delta(T_2)) \\ & \Leftrightarrow & \Delta \vdash \varepsilon_1 \sqsubseteq \alpha_\Delta(\{T_1\} \cap \{T_2\}) \\ & \Leftrightarrow & \Delta \vdash \varepsilon_1 \sqsubseteq \alpha_\Delta(\{T_2\}) \\ & \Leftrightarrow & \Delta \vdash \varepsilon_1 \sqsubseteq T_2 \\ & \Leftrightarrow & \varepsilon_1 = T_2 \end{array}$$

In a similar way $\varepsilon_2 = T_2$. Then $\varepsilon_1 = \varepsilon_2$.
Therefore, by the definition of consistent transitivity

$$\begin{array}{rcl} \varepsilon_1 \circ \varepsilon_2 & = & \alpha_\Delta(\gamma_\Delta(\varepsilon_1) \cap \gamma_\Delta(\varepsilon_2)) \\ & = & \alpha_\Delta(\gamma_\Delta(\varepsilon_1) \cap \gamma_\Delta(\varepsilon_1)) \\ & = & \alpha_\Delta(\gamma_\Delta(\varepsilon_1)) \\ & = & \alpha_\Delta(\{\varepsilon_1\}) \\ & = & \varepsilon_1 \end{array}$$

And the result holds. □

Lemma C.16. *Let* Δ *be a static context, then* $\widetilde{\mathrm{cty}}_\Delta(c) = \mathrm{cty}_\Delta(c)$*, if defined.*

Proof. Since the context is static the definition of $\widetilde{\mathsf{cty}_\Delta}(c)$ is:

$$\begin{array}{rcl}\widetilde{\mathsf{cty}_\Delta}(c) & = & \alpha_\Delta(\{\mathsf{cty}_\Delta(c) \mid \Delta_S \in \gamma(\Delta)\}) \\ & = & \alpha_\Delta(\{\mathsf{cty}_\Delta(c)\}) \\ & = & \mathsf{cty}_\Delta(c)\end{array}$$

And the result holds. □

Lemma C.17. *Let Δ and Ξ be static contexts, then $\widetilde{\mathsf{fty}_{\Delta,\Xi}}(l,T) = \mathsf{fty}_{\Delta,\Xi}(l,T)$, if defined.*

Proof. Since both contexts are static the definition of $\widetilde{\mathsf{fty}_{\Delta,\Xi}}(l,c)$ is:

$$\begin{array}{rcl}\widetilde{\mathsf{fty}_{\Delta,\Xi}}(l,T) & = & \alpha_\Delta(\{\mathsf{fty}_{\Delta,\Xi}(l,T) \mid \Delta_S \in \gamma(\Delta),\ \Xi_S \in \gamma_\Delta(\Xi), T' \in \gamma_\Delta(T)\}) \\ & = & \alpha_\Delta(\{\mathsf{fty}_{\Delta,\Xi}(l,T)\}) \\ & = & \mathsf{fty}_{\Delta,\Xi}(l,T)\end{array}$$

And the result holds. □

Lemma C.18. *Let Ξ be a static context, then $\widetilde{\mathsf{lty}_\Xi}(l,c) = \mathsf{lty}_\Xi(l,c)$, if defined.*

Proof. Since the context is static the definition of $\widetilde{\mathsf{lty}_\Xi}(l,c)$ is:

$$\begin{array}{rcl}\widetilde{\mathsf{lty}_\Xi}(l,c) & = & \alpha_\Delta(\{\mathsf{lty}_\Xi(l,c) \mid \Xi_S \in \gamma_\Delta(\Xi)\}) \\ & = & \alpha_\Delta(\{\mathsf{lty}_\Xi(l,c)\}) \\ & = & \mathsf{lty}_\Xi(l,c)\end{array}$$

And the result holds. □

Lemma C.19. *Let Δ and Ξ be static contexts and D a static datatype, then $\widetilde{\mathsf{valid}_{\Delta,\Xi}}(P,D) \Leftrightarrow \mathsf{valid}_{\Delta,\Xi}(P,D)$.*

Proof. Since the context is static the definition of $\widetilde{\mathsf{valid}_{\Delta,\Xi}}(P,D)$ is:

$$\begin{array}{rcl}\widetilde{\mathsf{valid}_{\Delta,\Xi}}(P,D) & = & \exists\Delta_S \in \gamma(\Delta), \exists\Xi_S \in \gamma_\Delta(\Xi), \exists D' \in \gamma_\Delta(D),\ \mathsf{valid}_{\Delta,\Xi}(P,D) \\ & = & \exists\Delta_S \in \{\Delta\}, \exists\Xi_S \in \{\Xi\}, \exists D' \in \{D\},\ \mathsf{valid}_{\Delta,\Xi}(P,D) \\ & = & \mathsf{valid}_{\Delta,\Xi}(P,D)\end{array}$$

And the result holds. □

Lemma C.20. *Let Δ and Ξ be static contexts, then $\widetilde{\mathsf{fty}_{\Delta,\Xi}}(l, \widetilde{\mathsf{cty}_\Delta}(c)) = \widetilde{\mathsf{lty}_\Xi}(l,c)$, if defined.*

Proof. By lemmas C.16 and C.17 we know that $\widetilde{\mathsf{fty}_{\Delta,\Xi}}(l, \widetilde{\mathsf{cty}_\Delta}(c)) = \mathsf{fty}_{\Delta,\Xi}(l, \mathsf{cty}_\Delta(c))$. By definition of cty_Δ, $c \in \mathsf{ctors}_\Delta(\mathsf{cty}_\Delta(c))$. Also by lemma C.18 we know that $\widetilde{\mathsf{lty}_\Xi}(l,c) = \mathsf{lty}_\Xi(l,c)$. Finally by definition of $\mathsf{fty}_{\Delta,\Xi}$, we have that $\mathsf{fty}_{\Delta,\Xi}(l, \mathsf{cty}_\Delta(c)) = \mathsf{lty}_\Xi(l,c')$ for any $c' \in \mathsf{ctors}_\Delta(\mathsf{cty}_\Delta(c))$. And the result holds. □

Lemma C.21 (Static preservation for $\longrightarrow$). *If Δ, Ξ and Γ are static contexts, $\Delta;\Xi;\Gamma \vdash e : T$ with e static and $e \longmapsto e'$ then e' is static.*

Proof. By induction on the derivation of $e \longmapsto e'$.

Case (εR-AscErase)

We have $e = \varepsilon_2\ (\varepsilon_1\ u :: T_1) :: T$. Then, by εR-AscErase

$$\varepsilon_2\ (\varepsilon_1\ u :: T_1) :: T \longrightarrow (\varepsilon_1 \circ \varepsilon_2)\ u :: T$$

with $(\varepsilon_1 \circ \varepsilon_2)$ defined.

Since ε_1 and ε_2 are static evidences, $(\varepsilon_1 \circ \varepsilon_2)$ is also static by lemma C.14.

Therefore $(\varepsilon_1 \circ \varepsilon_2)\ u :: T$ is static and the result holds.

Case (εR-Access)

We have $e = \varepsilon\ c\ \left\{\bar{l} = \overline{v}\right\} :: T_0.l$. Then, by εR-Access

$$\varepsilon\ c\ \left\{\bar{l} = \overline{v}\right\} :: T_0.l \longrightarrow (\varepsilon_k \circ \varepsilon_T)\ u_k :: T$$

with $l = l_k$, $v_k \doteq \varepsilon_k\ u_k :: T_k$, $T \doteq \widetilde{\mathsf{fty}_{\Delta,\Xi}}(l, T_0)$ and $(\varepsilon_k \circ \varepsilon_T)$ defined.

By lemma C.17, we know that T is a static type.

Since ε_k and ε_T are static evidences, $(\varepsilon_k \circ \varepsilon_T)$ is also static by lemma C.14.

Therefore $(\varepsilon_k \circ \varepsilon_T)\ u_k :: T$ is static and the result holds.

Case (εR-Match)

We have $e = \mathsf{match}\ \varepsilon\ c\ \left\{\bar{l} = \overline{v}\right\} :: T_0\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\}$. Then, by εR-Match

$$\mathsf{match}\ \varepsilon\ c\ \left\{\bar{l} = \overline{v}\right\} :: T_0\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\} \longrightarrow e_k\ [\overline{v}/\overline{x_k}]$$

with k the smallest integer such that $c = c_k$ where $c_k\ \overline{x_k} \doteq p_k$.

Since $\overline{v}$ and e_k are static, then $e_k\ [\overline{v}/\overline{x_k}]$ is static and the result holds.

□

Lemma C.22 (Static preservation). *If Δ, Ξ and Γ are static contexts, $\Delta; \Xi; \Gamma \vdash e : T$ with e static and $e \longmapsto e'$ then e' is static.*

Proof. By induction on the derivation of $e \longmapsto e'$.

Case (R $\longrightarrow$) Then

$$\frac{e \longrightarrow e'}{e \longmapsto e'}\ (\text{R} \longrightarrow)$$

and the result holds immediately by lemma C.21.

Case (RE) Then $e = E[e_0]$ and we have

$$\frac{e_0 \longmapsto e_0'}{E[e_0] \longmapsto E[e_0']} \text{ (RE)}$$

Since $\Delta; \Xi; \Gamma \vdash E[e_0] : T$ we know that $\Delta; \Xi; \Gamma \vdash e_0 : T_0$ with $E : T_0 \rightarrow T$.

By induction hypothesis we knwo that e_0' is static, since e_0 is static.

Therefore $E[e_0']$ is static and the result holds.

□

Lemma C.23 (Static progress). *When using matching strategy $m \in \{Sound, Exact, Complete\}$, if Δ, Ξ and Γ are static contexts, $\Delta; \Xi; \Gamma \vdash e : T$ with e static , then either*

- *e is a value*
- *$\exists e'$ s.t. $e \longmapsto e'$*
- *$e \longmapsto$ **err** with **err** $\in$ errors(m)*

Proof. By induction on the derivation of $\Delta; \Xi; \Gamma \vdash e : T$.

Case (εG-Ascribe)

We know that $e = \varepsilon\ e_0 :: T$. Then

$$\frac{\Delta; \Xi; \Gamma \vdash e_0 : T_0 \quad \varepsilon \Vdash T_0 = T}{\Delta; \Xi; \Gamma \vdash \varepsilon\ e_0 :: T : T} \text{ (εG-Ascribe)}$$

with e_0 static.

By induction hypothesis, we have either e_0 is a value, $e_0 \longmapsto e_0'$, or $e_0 \longmapsto$ **err** with **err** $\in$ errors(m).

- If e_0 is a value, then $e_0 \doteq \varepsilon_0\ u :: T_0$. Then

$$\frac{\Delta; \Xi; \Gamma \vdash u : T_u \quad \varepsilon_0 \Vdash T_u = T_0}{\Delta; \Xi; \Gamma \vdash \varepsilon_0\ u :: T_0 : T_0} \text{ (εG-Ascribe)}$$

  Since T, T_0 and T_u are static types, then $(\varepsilon_0 \circ \varepsilon)$ is defined and static by lemma C.15.

  Then, by εR-AscErase and εR $\longrightarrow$

$$\varepsilon\ (\varepsilon_0\ u :: T_0) :: T \longmapsto (\varepsilon_0 \circ \varepsilon)\ u :: T$$

  with $(\varepsilon_0 \circ \varepsilon)\ u :: T$ static.

- If $e_0 \longmapsto e_0'$, with e_0' static. Then $\varepsilon\ e_0 :: T \longmapsto \varepsilon\ e_0' :: T$ with $\varepsilon\ e_0' :: T$ static by εRE.
- If $e_0 \longmapsto \mathbf{err}$ with $\mathbf{err} \in \text{errors}(m)$, then $\varepsilon\ e_0 :: T \longmapsto \mathbf{err}$ with $\mathbf{err} \in \text{errors}(m)$ by εRErr.

And the result holds.

Case (εG-Ctor)

We know that $e = c\ \left\{\bar{l} = \overline{e}\right\}$. Then

$$\frac{\Delta;\Xi;\Gamma \vdash \overline{e} : \widetilde{\mathsf{lty}_\Xi}(\bar{l}, c) \quad \mathsf{satisfylabels}_{n\Xi}(c, l_1 \times \cdots \times l_n)}{\Delta;\Xi;\Gamma \vdash c\ \left\{\bar{l} = \overline{e}\right\} : T}\ (\varepsilon\text{G-Ctor})$$

where $\overline{e}$ are static.

By induction hypothesis, either every e_i is a value, $e_k \longmapsto e_k'$ with e_k' static and $\forall i < k.e_i$ is a value or $e_k \longmapsto \mathbf{err}$ with $\mathbf{err} \in \text{errors}(m)$ and $\forall i < k.e_i$ is a value.

- If e_i is a value, then $c\ \left\{\bar{l} = \overline{e}\right\}$ is a value.
- If $e_k \longmapsto e_k'$ with e_k' static and $\forall i < k.e_i$ is a value, then

$$c\ \left\{\bar{l} = \overline{v}, l_k = e_k, \bar{l} = \overline{e}\right\} \longmapsto c\ \left\{\bar{l} = \overline{v}, l_k = e_k', \bar{l} = \overline{e}\right\}$$

  by εRE.

  With $c\ \left\{\bar{l} = \overline{v}, l_k = e_k', \bar{l} = \overline{e}\right\}$ static.
- If $e_k \longmapsto \mathbf{err}$ with $\mathbf{err} \in \text{errors}(m)$ and $\forall i < k.e_i$ is a value, then

$$c\ \left\{\bar{l} = \overline{v}, l_k = e_k, \bar{l} = \overline{e}\right\} \longmapsto \mathbf{err}$$

  by εRErr.

And the result holds.

Case (εG-Access)

We know that $e = e_0.l$. Then

$$\frac{\Delta;\Xi;\Gamma \vdash e_0 : T_0 \quad \Delta \vdash T_0 \sqsubseteq ?_{\mathsf{D}}}{\Delta;\Xi;\Gamma \vdash e_0.l : T}\ (\varepsilon\text{G-Access})$$

with e_0 a static term and $T \doteq \widetilde{\mathsf{fty}_{\Delta,\Xi}}(l, T_0)$ a static type by lemma C.17.

By induction hypothesis, either e_0 is a value, $e_0 \longmapsto e_0'$ with e_0 static, or $e_0 \longmapsto \mathbf{err}$ with $\mathbf{err} \in \text{errors}(m)$.

- If e_0 is a value, then $e_0 \doteq \varepsilon\ c\ \left\{\bar{l} = \overline{v}\right\} :: T_0$ by Canonical Forms (Lemma C.1) since $\Delta \vdash T_0 \sqsubseteq ?_{\mathrm{D}}$.

  Then

$$\dfrac{\Delta; \Xi; \Gamma \vdash \overline{v} : \widetilde{\mathsf{lty}_{\Xi}}(\bar{l}, c) \quad \mathsf{satisfylabels}_{n\Xi}(c, l_1 \times \cdots \times l_n)}{\Delta; \Xi; \Gamma \vdash c\ \left\{\bar{l} = \overline{v}\right\} : T_0}\ (\varepsilon\text{G-Ctor})$$

  with $T_0 \doteq \widetilde{\mathsf{cty}_{\Delta}}(c)$.

  Either $l = l_k$ for some k or $\nexists l_k.l = l_k$.

  - If $l = l_k$ for some k, we have $v_k \doteq \varepsilon_k\ u_k :: T_k$, with $T_k \doteq \widetilde{\mathsf{lty}_{\Xi}}(l_k, c)$

    Then

$$\dfrac{\Delta; \Xi; \Gamma \vdash u_k : T_u \quad \varepsilon_k \Vdash T_u = T_k}{\Delta; \Xi; \Gamma \vdash \varepsilon_k\ u_k :: T_k : T_k}\ (\varepsilon\text{G-Ascribe})$$

    We know that $T_k = T$ by lemma C.20.

    Therefore $(\varepsilon_k \circ \varepsilon_T)$ is defined and static by lemma C.15.

    Then by εR-Access and εRE

$$\varepsilon\ c\ \left\{\bar{l} = \overline{v}\right\} :: T_0.l \longmapsto (\varepsilon_k \circ \varepsilon_T)\ u_k :: T$$

    with $(\varepsilon_k \circ \varepsilon_T)\ u_k :: T$ static.

  - If $\nexists l_k, l = l_k$, then by εR-Access and εRerr

$$\varepsilon\ c\ \left\{\bar{l} = \overline{v}\right\} :: T_0.l \longmapsto \mathbf{error_A}$$

  and $\mathbf{error_A} \in \mathrm{errors}(m)$ for every matching strategy.

- If $e_0 \longmapsto e_0'$ with e_0' static, then $e_0.l \longmapsto e_0'.l$ by εRE with $e_0'.l$ static.
- If $e_0 \longmapsto \mathbf{err}$ with $\mathbf{err} \in \mathrm{errors}(m)$, then $e_0.l \longmapsto \mathbf{err}$ by εRErr.

And the result holds.

Case (εG-Match)

We know that $e = \mathsf{match}\ e_0\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\}$. Then

$$\dfrac{\Delta; \Xi; \Gamma \vdash e_0 : T_0 \quad \begin{array}{c}\Delta \vdash T_0 \sqsubseteq ?_{\mathrm{D}} \\ \widetilde{\mathsf{valid}}_{\Delta,\Xi}(\{\overline{p}\}, T_0)\end{array} \quad \begin{array}{c}\forall i.1 \leq i \leq n \\ (x_{i1} : T_{i1}) \times \cdots \times (x_{im_i} : T_{im_i}) \doteq \widetilde{\mathsf{parg}_{m_i\Xi}}(p_i) \\ \Delta; \Xi; \Gamma, x_{i1} : T_{i1}, \ldots, x_{im_i} : T_{im_i} \vdash e_i : T\end{array}}{\Delta; \Xi; \Gamma \vdash \mathsf{match}\ e_0\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\} : T}\ (\varepsilon\text{G-Match})$$

with e_0 and $\overline{e}$ static.

By induction hypothesis, either e_0 is a value, $e_0 \longmapsto e_0'$ with e_0' static, or $e_0 \longmapsto \textbf{err}$ with $\textbf{err} \in \text{errors}(m)$.

- If e_0 is a value, then $e_0 \doteq \varepsilon\ c\ \left\{\overline{l} = \overline{v}\right\} :: T_0$ by Canonical Forms (Lemma C.1) since $\Delta \vdash T_0 \sqsubseteq ?_D$.

  Then

$$\dfrac{\dfrac{\Delta;\Xi;\Gamma \vdash \overline{v} : \widetilde{\text{lty}_{\Xi}}(\overline{l}, c) \quad \text{satisfylabels}_{n\Xi}(c, l_1 \times \cdots \times l_n)}{\Delta;\Xi;\Gamma \vdash c\ \left\{\overline{l} = \overline{v}\right\} : T_0'}\ (\varepsilon\text{G-Ctor}) \qquad \varepsilon \Vdash T_0' = T_0}{\Delta;\Xi;\Gamma \vdash \varepsilon\ c\ \left\{\overline{l} = \overline{v}\right\} :: T_0 : T_0}\ (\varepsilon\text{G-Ascribe})$$

  with $T_0 \doteq \widetilde{\text{cty}_{\Delta}}(c)$. Therefore $T_0 = D$ or some $D \in \text{dom}(\Delta)$. Since e_0 is well-typed, then $c \in ctors(D)$.

  Since every context is static and D is static, then $\widetilde{\text{valid}_{\Delta,\Xi}}(P, D) \iff \text{valid}_{\Delta,\Xi}(P, D)$ by lemma C.19.

  By an analisys of cases on m.

  – If $m = \text{Sound}$, then $\text{valid}_{\Delta,\Xi}(\left\{\overline{p}\right\}, D) \iff \bigcup(\text{pctor}(p_i)) \subseteq \text{ctors}_{\Delta}(D)$.

  If $c \in \bigcup(\text{pctor}(p_i))$, then $c = c_k$ with $c_k\overline{x_k} \doteq p_k$ for some p_k. Then by εR-Match and εRE

$$\text{match } c\ \left\{\overline{l} = \overline{v}\right\} \text{ with } \{\overline{p} \mapsto \overline{e}\} \longmapsto e_k\,[\overline{v}/\overline{x_k}]$$

  and the result holds.

  In contrast, if $c \notin \bigcup(\text{pctor}(p_i))$, then there is no k s.t. $c = c_k$ with $c_k\overline{x_k} \doteq p_k$. Then by εRerr

$$\text{match } c\ \left\{\overline{l} = \overline{v}\right\} \text{ with } \{\overline{p} \mapsto \overline{e}\} \longmapsto \textbf{error}_{\mathbf{M}}$$

  and since $\textbf{error}_{\mathbf{M}} \in \text{errors(Sound)}$ the result holds.

  – If $m = \text{Exact}$, then $\text{valid}_{\Delta,\Xi}(\left\{\overline{p}\right\}, D) \iff \bigcup(\text{pctor}(p_i)) = \text{ctors}_{\Delta}(D)$. Then it follows that $c \in \bigcup(\text{pctor}(p_i))$, and therefore $c = c_k$ with $c_k\overline{x_k} \doteq p_k$ for some p_k. Then by εR-Match and εRE

$$\text{match } c\ \left\{\overline{l} = \overline{v}\right\} \text{ with } \{\overline{p} \mapsto \overline{e}\} \longmapsto e_k\,[\overline{v}/\overline{x_k}]$$

  and the result holds.

  – If $m = \text{Complete}$, then $\text{valid}_{\Delta,\Xi}(\left\{\overline{p}\right\}, D) \iff \bigcup(\text{pctor}(p_i)) \supseteq \text{ctors}_{\Delta}(D)$. Then it follows that $c \in \bigcup(\text{pctor}(p_i))$, and therefore $c = c_k$ with $c_k\overline{x_k} \doteq p_k$ for some p_k. Then by

εR-MATCH and εRE

$$\text{match } c \ \left\{\overline{l} = \overline{v}\right\} \text{ with } \{\overline{p} \mapsto \overline{e}\} \longmapsto e_k \left[\overline{v}/\overline{x_k}\right]$$

and the result holds.

- If $e_0 \longmapsto e_0'$ with e_0' static, then

$$\text{match } e_0 \text{ with } \{\overline{p} \mapsto \overline{e}\} \longmapsto \text{match } e_0' \text{ with } \{\overline{p} \mapsto \overline{e}\}$$

by εRE with match e_0' with $\{\overline{p} \mapsto \overline{e}\}$ static.

- If $e_0 \longmapsto \mathbf{err}$ with $\mathbf{err} \in \text{errors}(m)$, then

$$\text{match } e_0 \text{ with } \{\overline{p} \mapsto \overline{e}\} \longmapsto \mathbf{err}$$

by εRERR.

And the result holds.

□

THEOREM 4.5 (STATIC TERMS DO NOT FAIL MORE). *Let e be a static term, T a static type, and Δ and Ξ static contexts. If $\Delta; \Xi; \cdot \vdash e : T$ and $e \Downarrow \mathbf{err}$, then $\mathbf{err} \in \text{errors}(m)$.*

PROOF. Direct by Preservation, Static Preservation, and Static Progress (Lemmas C.10, C.22 and C.23). □

C.5 Translation from $\lambda_{D?}$ to $\lambda^{\varepsilon}_{D?}$

In this section we present the translation from $\lambda_{D?}$ to $\lambda^{\varepsilon}_{D?}$ (Figure 26), which inserts ascriptions to every raw value and to every term whose typing rule makes a consistent judgment, synthetizing the corresponding evidence that supports the ascription. In this way all terms, except for ascriptions, have the exact type needed—and so we isolate possible inconsistencies to occur only in ascriptions. We use the following metafunctions to insert the ascriptions:

$$\begin{array}{rcl} \text{ascribe}_\Delta & : & \text{Expr} \times \text{GType} \times \text{GType} \rightharpoonup \text{Expr} \\ \text{ascribe}_\Delta(e, G_1, G_2) & = & \varepsilon e :: G_2 \quad \text{with } \varepsilon = \text{initial}_\Delta(G_1, G_2) \end{array}$$

$$\begin{array}{rcl} \text{initial}_\Delta & : & \text{GType} \times \text{GType} \rightharpoonup \text{GType} \\ \text{initial}_\Delta(G_1, G_2) & = & \alpha_\Delta(\gamma_\Delta(G_1) \cap \gamma_\Delta(G_2)) \end{array}$$

$$\begin{array}{rcl} \text{todata}_\Delta & : & \text{GType} \rightharpoonup \text{GType} \\ \text{todata}_\Delta(G) & = & \begin{cases} G & \Delta \vdash G \sqsubseteq ?_D \\ ?_D & G = ? \\ \bot & \text{otherwise} \end{cases} \end{array}$$

Translate $\boxed{\Delta; \Xi; \Gamma \vdash e \rightsquigarrow e_\varepsilon : G}$

$$\frac{G \doteq \Gamma(x) \qquad \Delta; \Xi \vdash \Gamma}{\Delta; \Xi; \Gamma \vdash x \rightsquigarrow x : G} \text{ Tr-Var} \qquad \frac{B \doteq \mathsf{ty}(k) \qquad \Delta; \Xi \vdash \Gamma}{\Delta; \Xi; \Gamma \vdash k \rightsquigarrow k : B} \text{ Tr-Const}$$

$$\frac{\Delta; \Xi; \Gamma; x : G_x \vdash e \rightsquigarrow e' : G}{\Delta; \Xi; \Gamma \vdash (\lambda x{:}G_x.\, e) \rightsquigarrow (\lambda x{:}G_x.\, e') : G_x \to G} \text{ Tr-Lam}$$

$$\frac{\Delta; \Xi; \Gamma \vdash u \rightsquigarrow u' : G \qquad \Delta; \Xi; \Gamma \vdash u' :: G \rightsquigarrow v : G}{\Delta; \Xi; \Gamma \vdash u \rightsquigarrow v : G} \text{ Tr-Value}$$

$$\frac{\begin{array}{c}\Delta; \Xi; \Gamma \vdash e_1 \rightsquigarrow e_1' : G_1 \qquad \Delta; \Xi; \Gamma \vdash e_2 \rightsquigarrow e_2' : G_2 \\ e_1'' \doteq \mathsf{ascribe}_\Delta(e_1', G_1, \widetilde{\mathsf{dom}}_\Delta(G_1) \to \widetilde{\mathsf{cod}}_\Delta(G_1)) \qquad e_2'' \doteq \mathsf{ascribe}_\Delta(e_2', G_2, \widetilde{\mathsf{dom}}_\Delta(G_1))\end{array}}{\Delta; \Xi; \Gamma \vdash e_1\ e_2 \rightsquigarrow e_1''\ e_2'' : \widetilde{\mathsf{cod}}_\Delta(G_1)} \text{ Tr-App}$$

$$\frac{\Delta; \Xi; \Gamma \vdash e \rightsquigarrow e' : G \qquad \varepsilon \doteq \mathsf{initial}_\Delta(G, G')}{\Delta; \Xi; \Gamma \vdash e :: G' \rightsquigarrow \varepsilon\, e' :: G' : G'} \text{ Tr-Asc}$$

$$\frac{\Delta; \Xi; \Gamma \vdash \overline{e} \rightsquigarrow \overline{e'} : \overline{G} \qquad \overline{e''} \doteq \mathsf{ascribe}_\Delta(\overline{e'}, \overline{G}, \widetilde{\mathsf{lty}_\Xi}(\overline{l}, c))}{\Delta; \Xi; \Gamma \vdash c\ \left\{\overline{l = e}\right\} \rightsquigarrow c\ \left\{\overline{l = e''}\right\} : G} \text{ Tr-Ctor}$$

$$\frac{\Delta; \Xi; \Gamma \vdash e \rightsquigarrow e' : G' \qquad e'' \doteq \mathsf{ascribe}_\Delta(e', G', \mathsf{todata}_\Delta(G'))}{\Delta; \Xi; \Gamma \vdash e.l \rightsquigarrow e''.l : G} \text{ Tr-Access}$$

$$\frac{\begin{array}{c}\Delta; \Xi; \Gamma \vdash e \rightsquigarrow e' : G' \qquad e'' \doteq \mathsf{ascribe}_\Delta(e', G', \mathsf{todata}_\Delta(G')) \\ (\forall i.\ 1 \le i \le n) \qquad (x_{i1} : G_{i1}) \times \cdots \times (x_{im_i} : G_{im_i}) \doteq \mathsf{parg}_{m_i \Xi}(p_i) \\ \Delta; \Xi; \Gamma; x_{i1} : G_{i1}, \ldots, x_{im_i} : G_{im_i} \vdash e_i \rightsquigarrow e_i' : G_i \\ G \doteq \widetilde{\mathsf{equate}}_{n\Delta}(G_1 \times \cdots \times G_n) \qquad e_i'' \doteq \mathsf{ascribe}_\Delta(e_i', G_i, G)\end{array}}{\Delta; \Xi; \Gamma \vdash \mathsf{match}\ e\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\} \rightsquigarrow \mathsf{match}\ e''\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e''}\} : G} \text{ Tr-Match}$$

Fig. 26. Translation from $\lambda_{\mathrm{D?}}$ to $\lambda^{\varepsilon}_{\mathrm{D?}}$

Theorem C.24 (Translation preserves typing). *If* $\Delta; \Xi; \Gamma \vdash e : G$, *then* $\Delta; \Xi; \Gamma \vdash e \rightsquigarrow e' : G$ *and* $\Delta; \Xi; \Gamma \vdash e' : G$.

Proof. By induction on the typing derivation $\Delta; \Xi; \Gamma \vdash e : G$.

Case (G-Ascribe)

Then $e = e_0 :: G$. Then

$$\frac{\Delta; \Xi; \Gamma \vdash e_0 : G_0 \quad \Delta \vdash G_0 \sim G}{\Delta; \Xi; \Gamma \vdash e_0 :: G : G} \text{ (G-Ascribe)}$$

By induction hypothesis $\Delta; \Xi; \Gamma \vdash e_0 \rightsquigarrow e_0' : G_0$ and $\Delta; \Xi; \Gamma \vdash e_0' : G_0$.

Let $\varepsilon = \mathsf{initial}_\Delta(G_0, G)$. Then $\varepsilon \Vdash \Delta \vdash G_0 \sim G$ by lemma C.3. Therefore $\Delta; \Xi; \Gamma \vdash e_0 :: G \rightsquigarrow \varepsilon\, e_0' :: G : G$ and $\Delta; \Xi; \Gamma \vdash \varepsilon\, e' :: G : G$ and the result holds.

Case (G-Ctor)

Then $e = c\ \left\{\overline{l} = \overline{e}\right\}$. Then

$$\frac{G \doteq \widetilde{\mathsf{cty}_\Delta}(c) \quad \Delta \vdash G \sim ?_{\mathrm{D}} \quad \Delta; \Xi; \Gamma \vdash \overline{e} : \overline{G} \quad \Delta \vdash \overline{G} \sim \widetilde{\mathsf{lty}_\Xi}(\overline{l}, c) \quad \mathsf{satisfylabels}_{n\Xi}(c, l_1 \times \cdots \times l_n)}{\Delta; \Xi; \Gamma \vdash c\ \left\{\overline{l} = \overline{e}\right\} : G}\ \text{(G-Ctor)}$$

By induction hypothesis $\Delta; \Xi; \Gamma \vdash \overline{e} \rightsquigarrow \overline{e'} : \overline{G}$ and $\Delta; \Xi; \Gamma \vdash \overline{e'} : \overline{G}$.

Then $\overline{e''} \doteq \mathsf{ascribe}_\Delta(\overline{e'}, \overline{G}, \widetilde{\mathsf{lty}_\Xi}(\overline{l}, c))$ with $\Delta; \Xi; \Gamma \vdash \overline{e''} : \widetilde{\mathsf{lty}_\Xi}(\overline{l}, c)$ by lemma C.4.

Therefore $\Delta; \Xi; \Gamma \vdash c\ \left\{\overline{l} = \overline{e}\right\} \rightsquigarrow c\ \left\{\overline{l} = \overline{e''}\right\} : G$ with $\Delta; \Xi; \Gamma \vdash c\ \left\{\overline{l} = \overline{e''}\right\} : G$ and the result holds.

Case (G-Access)

Then $e = e_0.l$. Then

$$\frac{\Delta; \Xi; \Gamma \vdash e_0 : G_0 \quad \Delta \vdash G_0 \sim ?_{\mathrm{D}}}{\Delta; \Xi; \Gamma \vdash e_0.l : \widetilde{\mathsf{fty}_{\Delta,\Xi}}(l, G_0)}\ \text{(G-Access)}$$

By induction hypothesis $\Delta; \Xi; \Gamma \vdash e_0 \rightsquigarrow e_0' : G_0$ and $\Delta; \Xi; \Gamma \vdash e_0' : G_0$.

Let $G_0' \doteq \mathsf{todata}_\Delta(G_0)$. We have $\Delta \vdash G_0' \sqsubseteq ?_{\mathrm{D}}$ by lemma C.7. Therefore $\Delta \vdash G_0 \sim ?_{\mathrm{D}}$.

Then $e_0'' = \mathsf{ascribe}_\Delta(e_0', G_0, G_0')$ with $\Delta; \Xi; \Gamma \vdash e_0'' : G_0'$ by lemma C.4.

Notice that $\widetilde{\mathsf{fty}_{\Delta,\Xi}}(l, G_0) = \widetilde{\mathsf{fty}_{\Delta,\Xi}}(l, G_0')$ by lemma C.6.

Therefore $\Delta; \Xi; \Gamma \vdash e_0.l \rightsquigarrow e_0''.l : \widetilde{\mathsf{fty}_{\Delta,\Xi}}(l, G_0)$ with $\Delta; \Xi; \Gamma \vdash e_0''.l : \widetilde{\mathsf{fty}_{\Delta,\Xi}}(l, G_0)$ and the result holds.

Case (G-Match)

Then $e = \mathsf{match}\ e_0\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\}$. Then

$$\frac{\Delta; \Xi; \Gamma \vdash e_0 : G_0 \quad \Delta \vdash G_0 \sim ?_{\mathrm{D}} \quad \widetilde{\mathsf{valid}_{\Delta,\Xi}}(\{\overline{p}\}, G_0) \quad \begin{array}{c} \forall i. 1 \leq i \leq n \\ (x_{i1} : G_{i1}) \times \cdots \times (x_{im_i} : G_{im_i}) \doteq \widetilde{\mathsf{parg}_{m_i\Xi}}(p_i) \\ \Delta; \Xi; \Gamma, x_{i1} : G_{i1}, \ldots, x_{im_i} : G_{im_i} \vdash e_i : G_i \end{array}}{\Delta; \Xi; \Gamma \vdash \mathsf{match}\ e_0\ \mathsf{with}\ \{\overline{p} \mapsto \overline{e}\} : \widetilde{\mathsf{equate}_{n\Delta}}(G_1, \ldots, G_n)}\ \text{(G-Match)}$$

By induction hypothesis $\Delta; \Xi; \Gamma \vdash e_0 \rightsquigarrow e_0' : G_0$ with $\Delta; \Xi; \Gamma \vdash e_0' : G_0$ and
$\Delta; \Xi; \Delta, x_{i1} : G_{i1},\ \ldots,\ x_{im_i} : G_{im_i} \vdash e_i \rightsquigarrow e_i' : G_i$ with
$\Delta; \Xi; \Gamma, x_{i1} : G_{i1},\ \ldots,\ x_{im_i} : G_{im_i} \vdash e_i' : G_i$.

Let $G_0' \doteq \text{todata}_\Delta(G_0)$. We have $\Delta \vdash G_0' \sqsubseteq ?_\mathsf{D}$ by lemma C.7. Therefore $\Delta \vdash G_0 \sim ?_\mathsf{D}$.

Then $e_0'' \doteq \text{ascribe}_\Delta(e_0', G_0, G_0')$ with $\Delta; \Xi; \Gamma \vdash e_0'' : G_0'$ and
$\overline{e''} \doteq \text{ascribe}_\Delta(\overline{e'}, \overline{G}, \widetilde{\text{equate}}_{n\Delta}(G_1 \times \cdots \times G_n))$ with $\Delta; \Xi; \Gamma \vdash \overline{e''} : \widetilde{\text{equate}}_{n\Delta}(G_1 \times \cdots \times G_n)$ by lemma C.4.

Also $\widetilde{\text{valid}}_{\Delta,\Xi}(\{\overline{p}\}, G_0')$ by lemma C.5.

Therefore $\Delta; \Xi; \Gamma \vdash \text{match } e_0 \text{ with } \{\overline{p} \mapsto \overline{e}\} \rightsquigarrow \text{match } e_0'' \text{ with } \{\overline{p} \mapsto \overline{e''}\} : G$ with $\Delta; \Xi; \Gamma \vdash$ $\text{match } e_0'' \text{ with } \{\overline{p} \mapsto \overline{e''}\} : G$ and the result holds.

$\square$