Label Dependent Lambda Calculus and Gradual Typing

WEILI FU, University of Freiburg, Germany

FABIAN KRAUSE, University of Freiburg, Germany

PETER THIEMANN, University of Freiburg, Germany

Dependently-typed programming languages are gaining importance, because they can guarantee a wide range of properties at compile time. Their use in practice is often hampered because programmers have to provide very precise types. Gradual typing is a means to vary the level of typing precision between program fragments and to transition smoothly towards more precisely typed programs. The combination of gradual typing and dependent types seems promising to promote the widespread use of dependent types.

We investigate a gradual version of a minimalist value-dependent lambda calculus. Compile-time calculations and thus dependencies are restricted to labels, drawn from a generic enumeration type. The calculus supports the usual Pi and Sigma types as well as singleton types and subtyping. It is sufficiently powerful to provide flexible encodings of variant and record types with first-class labels.

We provide type checking algorithms for the underlying label-dependent lambda calculus and its gradual extension. The gradual type checker drives the translation into a cast calculus, which extends the original language. The cast calculus comes with several innovations: refined typing for casts in the presence of singletons, type reduction in casts, and fully dependent Sigma types. Besides standard metatheoretical results, we establish the gradual guarantee for the gradual language.

CCS Concepts: • **Software and its engineering** → **Language features**; *Semantics*.

Additional Key Words and Phrases: dependent types, subtyping, gradual type systems

ACM Reference Format:

Weili Fu, Fabian Krause, and Peter Thiemann. 2021. Label Dependent Lambda Calculus and Gradual Typing. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 108 (October 2021), 29 pages. <https://doi.org/10.1145/3485485>

1 INTRODUCTION

Dependently-typed programming is on the rise. What started as a foundation for mathematics [Coquand and Huet 1988; Martin-Löf 1984] gave rise to powerful interactive verification tools [Bertot and Castéran 2004; Norell 2008] and is now influencing programming languages like Idris [Brady 2013], Haskell [Weirich et al. 2019], and TypeScript [typescript handbook 2021]. The trademark of dependent type theories is the dependent function type, where the type of the return value is calculated from the run-time value of the argument. While such calculations enable very precise compile-time reasoning, the high level of precision can make programming cumbersome and often requires extra effort to satisfy the type checker.

Gradual typing [Siek and Taha 2007, 2006] is all about relating program fragments that are typed at different levels of precision. In the original (extreme) case, gradual typing relates untyped and typed fragments, but many other flavors have been considered, for example situations where the

Authors' addresses: Weili Fu, University of Freiburg, Freiburg, Germany, weilifu@informatik.uni-freiburg.de; Fabian Krause, University of Freiburg, Freiburg, Germany, fabian.krause@students.uni-freiburg.de; Peter Thiemann, University of Freiburg, Freiburg, Germany, thiemann@acm.org.

This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2021 Copyright held by the owner/author(s).

2475-1421/2021/10-ART108

<https://doi.org/10.1145/3485485>

less typed fragment is the simply-typed lambda calculus and the more precisely typed fragment is some form of refinement type discipline [Knowles and Flanagan 2010; Ou et al. 2004; Wadler and Findler 2009]. A gradually typed program imposes the constraints of the more precisely typed fragment by performing run-time checks in the form of type casts at the fragment’s boundaries.

We study a gradual version of a dependently-typed system that is geared towards programming rather than specification and verification. The restricted scope enables us to consider subtyping and graduality together for the first time in a dependently-typed setting with full-fledged Σ -types.

1.1 Label Dependency and Subtyping

In a full-scale dependently-typed language like the calculus of constructions [Coquand and Huet 1988], there is a single syntactic category comprising types and terms. This feature enables all sorts of dependencies between types and terms, which greatly increases expressiveness. For example, the rule for dependent function application “imports” arbitrary argument terms into types. This freedom complicates type checking because comparing types is the same as comparing terms. The difficulty is that computation in the calculus is full (beta) reduction and results are normal forms. Hence, comparing two types/terms in the type checker requires reducing them to normal form, which must terminate to ensure decidability of type checking.

A value-dependent language restricts the terms that may be imported into types to values by requiring the arguments of dependent functions to be syntactic values. This choice simplifies the metatheory as well as type checking where evaluation (i.e., weak reduction instead of full reduction) is sufficient, but restricts the arguments passed to dependent functions to syntactic values.

Label-dependency goes one step further. A label-dependent calculus provides first-class label values ℓ , comparable to symbols in Lisp, along with an elimination construct **case** $V \{ \ell_1 : M_1, \dots \}$ that checks if V is one of the labels ℓ_j and branches to the selected term M_j . Any finite non-empty subset L of labels is a type in the calculus with subtyping generated by set inclusion of label sets. Each such L behaves like an extensible enumeration type without requiring an explicit declaration. The **case** on labels is the only operation that can be used at the type level.

The label-dependent lambda calculus, LDLC, which is our baseline in this work, is an extended subset of the calculus of label-dependent session types [Thiemann and Vasconcelos 2020]. It relies on a stratified syntax that distinguishes terms and types, which enables us to restrict the terms that are admissible in types: Types can only refer to label values with a type-level **case** construct where the branches are types; this **case** is a *large elimination* construct as it performs elimination of label values at the type level. Of course, there is a corresponding **case** construct at the term level, where the branches are terms. The term checked by **case** must be a syntactic value to avoid full normalization at the type level. As with value dependency, dependent functions must be applied to syntactic values.

As an example for LDLC consider a function *iors* that takes an argument of type **Bool** given by the two-element set of labels $\{T, F\}$ and returns an integer or a string:

$$\text{iors} = \lambda(x : \mathbf{Bool}) \mathbf{case} \ x \ \{ T : 42, F : \text{"bar"} \} : \Pi(x : \mathbf{Bool}) \mathbf{case} \ x \ \{ T : \mathbf{Int}, F : \mathbf{Str} \} \quad (1)$$

The **case** on the left is at the term level, whereas the **case** on the right is at the type level. Thanks to subtyping, we can apply this function to a value of type $\{T\}$ (which is a subtype of $\{T, F\} = \mathbf{Bool}$) and determine the typing $\text{iors } T : \mathbf{Int}$ at compile time. This term evaluates to 42.

We can also form a new label type on the fly and use a more general function like

$$\text{isd} = \lambda(x : \{T, F, D\}) \mathbf{case} \ x \ \{ T : 42, F : \text{"bar"}, D : 3.14159 \}$$

at a super type $\Pi(x : \mathbf{Bool})\mathbf{case} x \{T : \mathbf{Int}, F : \mathbf{Str}\}$. To apply *isd* (or *iors*) to a non-value like $(1 = 2) : \mathbf{Bool}$ or $(z = 2) : \mathbf{Bool}$ we have to let-abtract over the argument as follows:

$$\mathbf{let} x = (z = 2) \mathbf{in} (\mathbf{case} x \{T : \lambda x.x, F : \lambda x.\mathit{len}(x)\}) (\mathit{isd} x)$$

We call the subject of a type-level **case** an *index term* because it indexes the type with a run-time value that determines the type of the final result. In our example (1) the index term is the variable x .

1.2 Full-Fledged Sigma-Types

LDLC supports full-fledged Σ -types with call-by-value evaluation. An element of a Σ -type is a pair where the type of the second component depends on the value of the first component as in

$$\Sigma(x : \mathbf{Bool})\mathbf{case} x \{T : \mathbf{Int}, F : \mathbf{Str}\}$$

Typical terms of this type are $\langle x : \mathbf{Bool} = T, 42 \rangle$ and $\langle x : \mathbf{Bool} = F, \text{"foo"} \rangle$. Going beyond previous work, LDLC facilitates having the second component refer to the *value* of the first component as in

$$\lambda(z : \mathbf{Int})\langle x : \mathbf{Bool} = (z > 42), \mathbf{case} x \{T : z - 42, F : \mathit{str}(z)\} \rangle$$

This function has type $\mathbf{Int} \rightarrow \Sigma(x : \mathbf{Bool})\mathbf{case} x \{T : \mathbf{Int}, F : \mathbf{Str}\}$.

Generally, full Σ types allow for a flexible encoding of algebraic datatypes as well as giving expressive types to dictionaries in LDLC as explained further in Section 2.2. Value-dependent type systems with call-by-value evaluation (e.g. Swamy et al. [2013]) only support Σ -types where the term in the second component of a pair cannot refer to the first component. LDLC lifts this restriction, so that truly dependent records can be modeled by nested Σ -types.

1.3 Gradual Types

Gradual types [Siek and Taha 2007, 2006] enable the seamless transition between loosely typed code that makes use of the dynamic type $\star$ and statically-typed code. The idea is that explorative programming starts in a highly dynamic setting where most types are unknown. As the system takes shape, programmers gradually make the type structure more precise while exposing and fixing type mismatches. Ideally, this process continues until the final system is fully statically typed.

The gradual approach is advantageous if the underlying static type system offers a high degree of precision, which may make it hard to get types right from the beginning—in an exploratory programming effort, the code may not even survive to the next stage, so the effort to elaborate dynamic types may be wasted. Moreover, the gradual approach is called for if there are field-tested dynamically-typed libraries. Such libraries can be seamlessly integrated so they do not have to be migrated or upgraded to a fully typed version. Seamless means that statically-typed code executes with all static guarantees, there are checks at the boundaries from dynamically-typed code that establish those guarantees at run time, and the dynamically-typed code runs as is.

LDLC comes with a dependent type system that gives non-trivial guarantees and requires some degree of sophistication of the programmer. Its feature set is suited to model concepts like dictionaries in dynamically-typed languages using types. Thus, it is an ideal target for gradualization: to evolve types towards more precise descriptions and to interface with dynamically-typed code.

GLDLC (Gradual LDLC) enables the programmer to replace any *type* which is part of a type annotation in a program with the dynamic type. Generally, programs with dynamic type annotations are more often accepted by the gradual type checker, but may fail at run time if the types do not work out. For instance, GLDLC accepts all conceivable gradual variations of the examples as shown in Figure 1.¹ There is no type of the form **case** $\star \{ \dots \}$ because of stratification: the **case** expects a term at this position, not a type. Hence, this use of the type $\star$ is ruled out.

¹The [bluish background](#) indicates gradually typed code.

$\begin{aligned} \text{iors}' &= \lambda(x : \star) \text{case } x \{ T : 42, F : \text{"bar"} \} \\ &: \Pi(x : \star) \text{case } x \{ T : \text{Int}, F : \text{Str} \} \\ &: \Pi(x : \star) \text{case } x \{ T : \star, F : \text{Str} \} \\ &: \Pi(x : \star) \text{case } x \{ T : \text{Int}, F : \star \} \\ &: \Pi(x : \star) \text{case } x \{ T : \star, F : \star \} \\ &: \Pi(x : \star) \star \\ &: \star \end{aligned}$	$\begin{aligned} dp' &= \langle x : \star = F, \text{"foo"} \rangle \\ &: \Sigma(x : \star) \text{case } x \{ T : \text{Int}, F : \text{Str} \} \\ &: \Sigma(x : \star) \text{case } x \{ T : \star, F : \text{Str} \} \\ &: \Sigma(x : \star) \text{case } x \{ T : \text{Int}, F : \star \} \\ &: \Sigma(x : \star) \text{case } x \{ T : \star, F : \star \} \\ &: \Sigma(x : \star) \star \\ &: \star \end{aligned}$
---	--

Fig. 1. Different gradual types for the examples

The combination of gradual types with dependent types and subtyping gives rise to a range of interesting applications like encoding and parsing variant and record types with first-class labels, which we explore further in Section 2.

1.4 Contributions

- We define the label-dependent lambda calculus LDLC with singleton types, (label) value-dependent types, and subtyping. The calculus includes full-fledged Σ -types with large elimination. Section 3 presents the calculus along with its sound and complete bidirectional type checking algorithm.
- We specify the cast calculus CCLDLC (Section 4), which adds type casts to LDLC and comes with two main innovations: a refined approach to typing casts in the presence of singleton types and an operational semantics that includes reduction in the types of casts (Section 4.2).
- We present the gradual language GLDLC with algorithms for gradual type checking and subtyping in Section 5. To state correctness and completeness of the subtyping algorithm, we employ a new axiomatic definition of consistent subtyping. Our stratified syntax ensures that the marker $\star$ for the dynamic type does not block reductions.
- We extend the gradual type checking algorithm to a translation from GLDLC to the cast calculus in Section 5.2.
- We establish the refined criteria for gradual typing along with some other metatheoretical results in Section 6. See Section 4.3 for the standard metatheory of CCLDLC.

Full definitions and proofs may be found in the appendix of the extended version of the paper². A Haskell implementation of the type checker for LDLC and GLDLC and an Agda formalization of the metatheory of the cast language CCLDLC are available as artifacts³.

2 MOTIVATION

In this section, we dive a bit deeper into the example presented in the introduction and give further examples that demonstrate the working of gradual LDLC. We color-code the terms of the gradual calculus in **bluish background** and leave terms of the cast calculus in black in this section.

Section 2.1 is rather technical, but it serves to introduce some concepts that make the subsequent material more accessible. Section 2.2 discusses interfacing with dynamically typed code, in particular, parsing of JSON into a typed dictionary. Section 2.4 gives a view of the underlying run-time language, the cast calculus. Section 2.5 discusses some intricacies that reappear in the technical material.

²<https://freidok.uni-freiburg.de/data/220838>

³<https://doi.org/10.5281/zenodo.5497628>

2.1 A Taste of the Gradual Label-Dependent Lambda Calculus

Our starting point, the label-dependent lambda calculus (LDLC), is inspired by the type system underlying label-dependent session types [Thiemann and Vasconcelos 2020]. It is a call-by-value lambda calculus with a special type of finite label sets (i.e., a generic enumeration type). Non-empty label sets can serve as index types to create value-dependent Π - and Σ -types. This way, LDLC can model records with label-dependent Π -types as well as variants with label-dependent Σ -types; both with first-class labels. Subtyping is generated from inclusion of label sets and singleton types.

As an example of an LDLC function, consider

$$f = \lambda(x : \mathbf{Bool}).\lambda(y : \mathbf{case } x \{T : \mathbf{Int}, F : \mathbf{Bool}\}).\mathbf{case } x \{T : 17 + y, F : \mathbf{not } y\} \\ : \Pi(x : \mathbf{Bool}).\Pi(y : \mathbf{case } x \{T : \mathbf{Int}, F : \mathbf{Bool}\}).\mathbf{case } x \{T : \mathbf{Int}, F : \mathbf{Bool}\}$$

where $\mathbf{Bool} = \{T, F\}$ abbreviates the label type inhabited by the labels T and F and $\mathbf{case}$ is the eliminator for label types. This function accepts a boolean x and returns a function that either takes an integer and returns an integer or it takes a boolean and returns a boolean, depending on x .

Let us consider two gradual versions of this function so as to highlight the most significant issues in gradualizing LDLC. As usual, the gradual language allows us to replace types in the type annotations by the dynamic type denoted by $*$.

$$f_1 = \lambda(x : \mathbf{Bool}).\lambda(y : *).\mathbf{case } x \{T : 17 + y, F : \mathbf{not } y\} \\ : \Pi(x : \mathbf{Bool}).\Pi(y : *).\mathbf{case } x \{T : \mathbf{Int}, F : \mathbf{Bool}\}$$

The function f_1 and its behavior is unsurprising for a gradual language. As y is dynamic, it can be used in a context expecting an $\mathbf{Int}$ as well as in one expecting a $\mathbf{Bool}$. If f_1 is applied to an unsuitable value for y , say a string, then we get an error at run time. This error is flagged by a cast that is inserted by elaborating the gradual language to a run-time language with type casts.

$$f_2 = \lambda(x : *).\lambda(y : \mathbf{case } x \{T : \mathbf{Int}, F : \mathbf{Bool}\}).\mathbf{case } x \{T : 17 + y, F : \mathbf{not } y\} \\ : \Pi(x : *).\Pi(y : \mathbf{case } x \{T : \mathbf{Int}, F : \mathbf{Bool}\}).\mathbf{case } x \{T : \mathbf{Int}, F : \mathbf{Bool}\}$$

The function f_2 is more interesting. Here the type of x is dynamic, but nevertheless the gradual type checker realizes that the binding of y and its use have the same type, under the condition that x is either T or F at run time. If we try to apply this function to a value of inappropriate type, as in $f_2()$, then this application is rejected because the type $\mathbf{case } x \{T : \mathbf{Int}, F : \mathbf{Bool}\}$ is ill-formed after substitution $[()/x]$. If we apply f_2 to a dynamic variable $z : *$, then we obtain a function of type

$$f_2 z : \Pi(y : \mathbf{case } z \{T : \mathbf{Int}, F : \mathbf{Bool}\}).\mathbf{case } z \{T : \mathbf{Int}, F : \mathbf{Bool}\}$$

which ought to be applicable to any value that has a chance to be acceptable as type $\mathbf{case } z \{T : \mathbf{Int}, F : \mathbf{Bool}\}$, that is, $*$, $\mathbf{Int}$, or $\mathbf{Bool}$ (among further possibilities involving types formed using $\mathbf{case}$). We say that each of $*$, $\mathbf{Int}$, and $\mathbf{Bool}$ is a *consistent subtype* of $\mathbf{case } z \{T : \mathbf{Int}, F : \mathbf{Bool}\}$.

2.2 Typed Records, Variants, and Dictionaries

All programming languages have means to express structures composed of subsidiary values using records, structs, objects, or dictionaries. Often, particular types are needed to express such structures, but in a dependently-typed language, the Π -type is sufficient to model them. As an example, a record describing a person can be represented in LDLC by a value of type $Person$ where we assume that labels are strings:

$$Person = \Pi(x : \{ "name", "age", "flag" \}).\mathbf{case } x \{ "name" : \mathbf{Str}, "age" : \mathbf{Int}, "flag" : \mathbf{Bool} \}$$

For convenience we informally use a hypothetical extension of LDLC with suitable base types like strings and various types of numbers in this subsection. We also freely allow any value to be used as a label, as long as the only operation applied to it is equality as part of the **case** distinction. None of these liberties violate the theoretical foundations.

In a dynamically-typed language like Python, a value of type *Person* could be represented by a dictionary as follows:

```
aPerson = { 'name': 'Masato_Honda', 'age': 58, 'flag': true }
```

Given a Python function `getDict` that returns a dictionary, we might import it to GLDLC using the type $\Pi(t : *)$. If we expect the dictionary to represent a person, then we can also use the returned value at type *Person*. If we apply this value to one of the labels from the type, then gradual typing ensures that the returned value has the type prescribed by *Person*. That is, if we query for "age", then we either get a run-time error or we are guaranteed to get a value of type **Int**.

In a similar way, we can integrate a JSON parser in GLDLC. When converting a JSON structure into a typed internal representation (i.e., a record), the converter knows about the expected labels and the types of the corresponding fields. A JSON value is a number, a string, a boolean, or an object, which maps labels to JSON values.⁴

The generic JSON parser would represent all JSON values using the dynamic type. An object value would be represented as a function of type $\Pi(t : *)$. Thanks to gradual typing, we can apply a function taking an argument of type *Person* directly to the dynamic output $d : *$ of the parser. If we expect an object, then we can apply d to a tag t . This application can fail in three ways, either if d is not an object, if d does not have a component tagged t , or if component t does not have the expected type. If the application succeeds, we can be sure to obtain a value of the correct type for t .

We can make the *Person* type more concise and more modular by using a gradual type instead:

$$Person' = \Pi(x : *) \text{case } x \{ "name" : \mathbf{Str}, "age" : \mathbf{Int}, "flag" : \mathbf{Bool} \}$$

Values of this type would behave just like values of type *Person*. If $p : Person$, then applying p to tag "bday" is a type error; likewise, if $p' : Person'$, then $p' "bday"$ is also a type error. We regard the type *Person'* as more modular because it does not force us to repeat the labels in the domain of the function. If we wish to add further "fields", it is sufficient to extend the **case** type with an additional alternative.

If we care about the presence of a tag, but not about its details, yet, we can write

$$Person'' = \Pi(x : *) \text{case } x \{ "name" : \mathbf{Str}, "parent" : *, \dots \}$$

Later on, we may try to use the "parent" field as a person. That is if $p'' : Person''$, then we can say $p'' "parent" "name"$, for example. Sadly, the ellipsis ... is not part of the syntax, but indicates cases that we omit for brevity. It would be interesting future work to consider an extension of GLDLC's **case**-types that supports defining a default return type if none of the explicit cases matches the given label value.

An analogous scheme can be used to represent variant types. We just have to put a Σ -type in place of the Π -type. We might express the converter as a function that takes a tagged value of type $\Sigma(t : *)$ where t is the tag and the second component is the associated value.

$$OptionalInt = \Sigma(t : \{ "none", "some" \}) \text{case } t \{ "none" : \mathbf{Unit}, "some" : \mathbf{Int} \}$$

There is also a gradual version in analogy to *Person'*:

⁴We omit arrays as they offer no new insights.

$$OptionalInt' = \Sigma(t : *) \text{case } t \{ \text{"none"} : \mathbf{Unit}, \text{"some"} : \mathbf{Int} \}$$

Again, we can apply a function with argument type *OptionalInt* directly to the dynamic output of the parser to obtain a typed optional value in the function body.

2.3 Flexible Mapping

In this subsection, we consider an example from [Siek and Vachharajani \[2008\]](#) that exemplifies the interplay between polymorphism and the dynamic type. They propose a flexible mapping function that either maps a function over the elements of an array or a list. Their input type is dynamic as the function chooses at run time between array and list input. We are using fixed types *A* and *B* as GLDLC does not support polymorphism.

$$mymap = \lambda(b : \mathbf{Bool}) \lambda(f : A \rightarrow B) \lambda(x : *) \text{case } b \{ T : \text{Array.map } f \ x, F : \text{List.map } f \ x \}$$

Assuming the typings $\text{Array.map} : (A \rightarrow B) \rightarrow (\text{Array } A \rightarrow \text{Array } B)$ and $\text{List.map} : (A \rightarrow B) \rightarrow (\text{List } A \rightarrow \text{List } B)$ we obtain the type

$$mymap : \Pi(b : \mathbf{Bool}) \Pi(f : A \rightarrow B) \Pi(x : *) \text{case } b \{ T : \text{Array } B, F : \text{List } B \}$$

We remark that this type does not exploit the expressive power of our calculus. In LDLC we are able to just give a fully static type to *mymap*:

$$\Pi(b : \mathbf{Bool}) \Pi(f : A \rightarrow B) \Pi(x : \text{case } b \{ T : \text{Array } A, F : \text{List } A \}) \text{case } b \{ T : \text{Array } B, F : \text{List } B \}$$

Moreover, we can still type check this definition in GLDLC if we make all arguments dynamic. While this type gives up some safety at run time, our type checker does not lose precision when inferring the return type.

$$dynmap : \Pi(b : *) \Pi(f : *) \Pi(x : *) \text{case } b \{ T : \text{Array } B, F : \text{List } B \}$$

[Castagna and Lanvin \[2017\]](#) discuss this example in the context of set-theoretic types where the return type of *mymap* is $\text{Array } B \vee \text{List } B$ and remark that this type is the same as the one returned by the following variation of *mymap*:

$$mymap2 = \lambda(b : \mathbf{Bool}) \lambda(f : A \rightarrow B) \lambda(x : *) \\ \text{case } b \{ T : \text{Array.to_list}(\text{Array.map } f \ x), F : \text{Array.of_list}(\text{List.map } f \ x) \}$$

Assuming the typings $\text{Array.to_list} : \text{Array } B \rightarrow \text{List } B$ and $\text{Array.of_list} : \text{List } B \rightarrow \text{Array } B$, we obtain the GLDLC typing:

$$mymap2 : \Pi(b : \mathbf{Bool}) \Pi(f : A \rightarrow B) \Pi(x : *) \text{case } b \{ T : \text{List } B, F : \text{Array } B \}$$

This definition has a statically-typed counterpart in LDLC if we assign *x* a suitable type:

$$x : \text{case } b \{ T : \text{Array } A, F : \text{List } A \}$$

2.4 A Taste of the Cast Calculus

Up to now, we considered terms at the level of the external language GLDLC. Before evaluating such a term, we must translate it to the internal language CCLDLC by inserting type casts of the form $(M : A \Rightarrow B)$. These cast mediate between static types and the dynamic type. They boil down to a run-time check that types *A* and *B* are consistent. Let's illustrate that with some examples.

The most basic casts have the form $G \Rightarrow *$ and $* \Rightarrow G$. Here, *G* is a *ground type* which indicates just the top-level type constructor. For example, **Unit** or any label type is a ground type. The ground type for Π -types is $\Pi(x : *) *$, a function with unknown argument and return type, and for Σ -types

it is $\Sigma(x : \star) \star$, a pair with unknown component types. At run time, any cast is decomposed down to basic casts. The cast $G \Rightarrow \star$ plays the role of a type tag and becomes part of a dynamic value. Such a value is inspected with the cast $\star \Rightarrow H$, which succeeds if G is a subtype of H and fails otherwise. The latter cast essentially checks a type tag and peels it off if the check succeeds.

These are the casts that we see at work in this section. The remaining casts transform function and pair types by propagating casts to the respective components. They are explained in Section 4.2.

Continuing with f_2 , it turns out that each use of x must be wrapped in a cast because it is expected to be used as a boolean. For non-dependent gradual languages, the type must not be translated, but in GLDLC the translation affects terms and types.

$$f_2' = \lambda(x : \star). \lambda(y : \text{case } (x : \star \Rightarrow \text{Bool}) \{T : \text{Int}, F : \text{Bool}\}). \text{case } (x : \star \Rightarrow \text{Bool}) \{T : 17 + y, F : \text{not } y\} \\ : \Pi(x : \star). \Pi(y : \text{case } (x : \star \Rightarrow \text{Bool}) \{T : \text{Int}, F : \text{Bool}\}). \text{case } (x : \star \Rightarrow \text{Bool}) \{T : \text{Int}, F : \text{Bool}\}$$

To type check this term, CCLDLC enters the branches of the term $\text{case } (x : \star \Rightarrow \text{Bool}) \{T : 17 + y, F : \text{not } y\}$ twice with a refined type assumption for x , namely $x : \mathbf{S}\{(T : \text{Bool} \Rightarrow \star) : \star\}$ and $x : \mathbf{S}\{(F : \text{Bool} \Rightarrow \star) : \star\}$, respectively. The subjects of these singleton types are particular values of type $\star$, so the types are subtypes of $\star$ and thus valid refinements of the assumed type of $x : \star$. When it comes to checking the type of y in the T -branch, we find that

$$\dots, x : \mathbf{S}\{(T : \text{Bool} \Rightarrow \star) : \star\}, \dots \vdash y : \text{case } (x : \star \Rightarrow \text{Bool}) \{T : \text{Int}, F : \text{Bool}\}$$

We claim that we can convert the type of y to **Int** using the assumption on x . How is that possible?

In Section 4, we define a refined rule for type checking type casts, which keeps the precision of incoming singleton types by attempting to apply the cast to the singleton value. With this rule we derive the type of the **case** header:

$$\dots, x : \mathbf{S}\{(T : \text{Bool} \Rightarrow \star) : \star\}, \dots \vdash (x : \star \Rightarrow \text{Bool}) : \mathbf{S}\{T : \text{Bool}\}$$

As the header of the **case**-type has a singleton type, the type of y is convertible to the selected branch, which is **Int**.

As another example, we continue our discussion of $f_2 z$ where $z : \star$. If $v : \text{Int}$ is the next argument, then at run time we will be looking at the following term. We insert the **let** for readability.

$$\text{let } y = (v : \text{Int} \Rightarrow \text{case } (z : \star \Rightarrow \text{Bool}) \{T : \text{Int}, F : \text{Bool}\}) \text{ in } f_2 z y \\ : \text{case } (z : \star \Rightarrow \text{Bool}) \{T : \text{Int}, F : \text{Bool}\}$$

Once the value of z is known, we must *reduce the type in the cast*. For example, if $z = (F : \text{Bool} \Rightarrow \star)$:

$$\text{let } y = (v : \text{Int} \Rightarrow \text{case } ((F : \text{Bool} \Rightarrow \star) : \star \Rightarrow \text{Bool}) \{T : \text{Int}, F : \text{Bool}\}) \text{ in } f_2 F y \\ \text{let } y = (v : \text{Int} \Rightarrow \text{case } F \{T : \text{Int}, F : \text{Bool}\}) \text{ in } f_2 F y \\ \longrightarrow \text{let } y = (v : \text{Int} \Rightarrow \text{Bool}) \text{ in } f_2 F y \\ : \text{case } ((F : \text{Bool} \Rightarrow \star) : \star \Rightarrow \text{Bool}) \{T : \text{Int}, F : \text{Bool}\}$$

In the next step, the cast $\text{Int} \Rightarrow \text{Bool}$ fails and stops execution. The type starts out as shown in the last line. It is convertible to **Bool**.

While this example does not exercise subtyping explicitly, it is actually omnipresent when dealing with label types. For instance, the application $f T$ already involves subtyping and singleton types as $T : \mathbf{S}\{T : \text{Bool}\}$ (the singleton type for value T) and $\mathbf{S}\{T : \text{Bool}\} <: \text{Bool}$, informally.

2.5 The Spectrum of Gradual LDLC

We already discussed that f_2 cannot take an argument of unsuitable type. But there is another way in which f_2 could fail: What if the two arguments x and y are “out of sync” as in $f_2 F 42$, where F

demands a boolean argument but an integer is supplied? To answer this question, we first translate $f_2 F$ to CCLDLC using $W = (F : \mathbf{Bool} \Rightarrow \star)$.

$$f'_2 W : \Pi(y : \mathbf{case} (W : \star \Rightarrow \mathbf{Bool}) \{T : \mathbf{Int}, F : \mathbf{Bool}\}).\mathbf{case} (W : \star \Rightarrow \mathbf{Bool}) \{T : \mathbf{Int}, F : \mathbf{Bool}\}$$

By virtue of singleton types and the refined cast rule, this type is convertible to $\Pi(y : \mathbf{Bool}).\mathbf{Bool}$. Hence, it is impossible to misuse f_2 once its arguments are sufficiently known.

However, we may consider the application $f_2 z 42$, where z is a free variable of type dynamic. Once again, we look at the translated term and its type.

$$\begin{aligned} \text{let } y = (42 : \mathbf{Int} \Rightarrow \mathbf{case} (z : \star \Rightarrow \mathbf{Bool}) \{T : \mathbf{Int}, F : \mathbf{Bool}\}) \text{ in } f'_2 z y \\ : \mathbf{case} (z : \star \Rightarrow \mathbf{Bool}) \{T : \mathbf{Int}, F : \mathbf{Bool}\} \end{aligned}$$

Suppose now a surrounding beta reduction substitutes $[(F : \mathbf{Bool} \Rightarrow \star)/z]$ leaving us with:

$$\begin{aligned} \text{let } y = (42 : \mathbf{Int} \Rightarrow \mathbf{case} ((F : \mathbf{Bool} \Rightarrow \star) : \star \Rightarrow \mathbf{Bool}) \{T : \mathbf{Int}, F : \mathbf{Bool}\}) \text{ in } f'_2 (F : \mathbf{Bool} \Rightarrow \star) y \\ : \mathbf{case} ((F : \mathbf{Bool} \Rightarrow \star) : \star \Rightarrow \mathbf{Bool}) \{T : \mathbf{Int}, F : \mathbf{Bool}\} \end{aligned}$$

To proceed, we must evaluate inside the type in the cast as indicated by the underlined redex. (The same reduction is also part of type conversion, so we keep the type synchronized even though the two notions of type reduction in a cast and type conversion are separate from each other.)

$$\begin{aligned} \text{let } y = (42 : \mathbf{Int} \Rightarrow \mathbf{case} F \{T : \mathbf{Int}, F : \mathbf{Bool}\}) \text{ in } f'_2 (F : \mathbf{Bool} \Rightarrow \star) y \\ : \mathbf{case} F \{T : \mathbf{Int}, F : \mathbf{Bool}\} \end{aligned}$$

Once again, we reduce the underlined type in the cast (and apply conversion in the type) to obtain:

$$\text{let } y = (42 : \mathbf{Int} \Rightarrow \mathbf{Bool}) \text{ in } f'_2 (F : \mathbf{Bool} \Rightarrow \star) y : \mathbf{Bool}$$

In the next step, the cast $\mathbf{Int} \Rightarrow \mathbf{Bool}$ fails, which indicates that the value of x was not in line with the type of y . This cast failure has a distinctly different flavor than cast failures in other gradual systems. Most systems require that source and target type of a cast are consistent so that there is always some unwrapping of a dynamic type involved in a failing cast.

Our translation in Section 5.2 inserts a cast for types A and B if A and B are *consistent subtypes*. Consistent subtyping succeeds if there is any way to refine the gradual types A and B and the types of values in the environment on which A or B depend, such that the resulting precise types are subtypes in LDLC. Clearly, as long as z is unknown, there is a chance that z becomes $(T : \mathbf{Bool} \Rightarrow \star)$, which would make $\mathbf{Int}$ a subtype of $\mathbf{case} (z : \star \Rightarrow \mathbf{Bool}) \{T : \mathbf{Int}, F : \mathbf{Bool}\}$. This possibility justifies the original insertion of the cast instead of rejecting the code outright. However, the consistent subtyping between source and target of a cast is not preserved under reduction, as the example demonstrates.

For yet another gradualization, we could make the F -branch of the $\mathbf{case}$ type dynamic.

$$\begin{aligned} f_3 = \lambda(x : \star).\lambda(y : \mathbf{case} x \{T : \mathbf{Int}, F : \star\}).\mathbf{case} x \{T : 17 + y, F : \mathbf{not} y\} \\ : \Pi(x : \star).\Pi(y : \mathbf{case} x \{T : \mathbf{Int}, F : \star\}).\mathbf{case} x \{T : \mathbf{Int}, F : \mathbf{Bool}\} \end{aligned}$$

At this gradual type, the application $f_3 F 42$ is well-typed in GLDLC. Again, we set $W = (F : \mathbf{Bool} \Rightarrow \star)$ and consider the translation

$$f'_3 W : \Pi(y : \mathbf{case} (W : \star \Rightarrow \mathbf{Bool}) \{T : \mathbf{Int}, F : \star\}).\mathbf{case} (W : \star \Rightarrow \mathbf{Bool}) \{T : \mathbf{Int}, F : \mathbf{Bool}\}$$

But now, type conversion yields

$$f'_3 W : \Pi(y : \mathbf{case} F \{T : \mathbf{Int}, F : \star\}).\mathbf{case} F \{T : \mathbf{Int}, F : \mathbf{Bool}\}$$

Terms	$M, N ::= x \mid () \mid \ell \mid \mathbf{case} X \{\overline{\ell : N_\ell}\} \mid \mathbf{let} x = M \mathbf{in} N$ $\mid \lambda(x : A).M \mid MN \mid \langle x : A = M, N \rangle \mid \langle\langle V, N \rangle\rangle \mid \mathbf{let} \langle x, y \rangle = M \mathbf{in} N$
Values	$V, W ::= () \mid \ell \mid \lambda(x : A).M \mid \langle\langle V, W \rangle\rangle$
Gen. Values	$X, Y ::= x \mid V$
Types	$A, B ::= \mathbf{Unit} \mid \mathbf{S}\{V : A\} \mid L \mid \Pi(x : A)B \mid \Sigma(x : A)B \mid \mathbf{case} X \{\overline{\ell : A_\ell}\}$
Label Types	$L ::= \{\ell_1, \dots, \ell_n\} \quad n > 0, (\forall i, j) 1 \leq i < j \leq n \Rightarrow \ell_i \neq \ell_j$

Fig. 2. Syntax of LDLC

which is applicable to 42 after applying the appropriate cast:

$$f'_3 W (42 : \mathbf{Int} \Rightarrow \underline{\mathbf{case} F \{T : \mathbf{Int}, F : \star\}}) : \mathbf{case} F \{T : \mathbf{Int}, F : \mathbf{Bool}\}$$

and type reduction

$$f'_3 W (42 : \mathbf{Int} \Rightarrow \star) : \mathbf{case} F \{T : \mathbf{Int}, F : \mathbf{Bool}\}$$

Finally, we beta reduce f'_3 to obtain

$$\mathbf{case} (W : \star \Rightarrow \mathbf{Bool}) \{T : \underline{17 + (42 : \mathbf{Int} \Rightarrow \star)}, F : \mathbf{not}((42 : \mathbf{Int} \Rightarrow \star) : \star \Rightarrow \mathbf{Bool})\} : \mathbf{Bool}$$

This term is typed despite the untypeable underlined subterm. The typing rule for **case** only checks the branches which are reachable according to the type of the header. In this case, the header $(W : \star \Rightarrow \mathbf{Bool})$ has type $\mathbf{S}\{F : \mathbf{Bool}\}$. Hence the T -branch is not checked.

Reducing the cast and the case yields $\mathbf{not}((42 : \mathbf{Int} \Rightarrow \star) : \star \Rightarrow \mathbf{Bool}) : \mathbf{Bool}$, which fails in the next reduction step.

3 THE LABEL DEPENDENT LAMBDA CALCULUS

LDLC is a dependently-typed lambda calculus where dependencies are restricted to labels. The dynamics of LDLC are mostly standard, but its statics are more involved. They are inspired by a range of earlier work, most notably Zombie/Trellys [Casinghino et al. 2014; Sjöberg et al. 2012] and F* [Swamy et al. 2013], to formalize a flexible dependently-typed system based on call-by-value execution. The treatment of dependent Σ -types is different from previous work.

Section 4 describes the cast calculus as a conservative extension of LDLC. We defer proofs of agreement (Lemma 4.4) and type soundness (Section 4.3) to that section.

3.1 Syntax and Dynamics

Figure 2 describes LDLC's syntax, comprising terms, values, generalized values, and types. Types comprise the unit type; label types L inhabited by finitely many elements of an underlying universe $\mathcal{L}$ of labels; the dependent function and product types $\Pi(x : A)B$ and $\Sigma(x : A)B$. The type $\mathbf{case} X \{\overline{\ell : A_\ell}\}$ eliminates labels and is restricted to generalized values X (a value or a variable). The singleton type $\mathbf{S}\{V : A\}$ has the value V as its only member, which must be of base type (i.e., **Unit** or L).

Terms of LDLC are variables, unit, labels, **case** expressions, typed abstractions, applications, and constructions for dependent pairs. In a dependent pair $\langle x : A = M, N \rangle$, the first component M is bound to a variable x which may be used in the second component. This arrangement results in a number of complications. Once the first component, M , is reduced to a value, the second component, N , may still refer to x as in $\langle x : L = \ell, \mathbf{case} x \{\dots\} \rangle$. To avoid that this expression

Evaluation contexts		$\mathcal{E}, \mathcal{F} ::= \square \mid \mathcal{E} N \mid V \mathcal{E} \mid \mathbf{let} x = \mathcal{E} \mathbf{in} N$ $\mid \langle x : A = \mathcal{E}, N \rangle \mid \langle \langle V, \mathcal{E} \rangle \rangle \mid \mathbf{let} \langle x, y \rangle = \mathcal{E} \mathbf{in} N$	
Term reduction		$M \longrightarrow N$	
RL-CASE	$\frac{\ell' \in \{\bar{\ell}\}}{\mathbf{case} \ell' \{ \ell : \bar{M}_{\ell} \} \longrightarrow M_{\ell'}}$	RL-BETAV	$(\lambda(x : A).M) V \longrightarrow M[V/x]$
		RL-PROD-ELIM	$\mathbf{let} \langle x, y \rangle = \langle \langle V, W \rangle \rangle \mathbf{in} N \longrightarrow N[V, W/x, y]$
RL-LETV	$\mathbf{let} x = V \mathbf{in} M \longrightarrow M[V/x]$	RL-PROD	$\langle x : A = V, N \rangle \longrightarrow \langle \langle V, N[V/x] \rangle \rangle$
		RL-CTX-EXP	$\frac{M \longrightarrow N}{\mathcal{E}[M] \longrightarrow \mathcal{E}[N]}$

Fig. 3. Reduction in LDLC

gets stuck, we introduce a novel reduction RL-PROD that substitutes the value for x in the second component. This reduction results in a resolved pair $\langle \ell, \mathbf{case} \ell \{ \dots \} \rangle$, where the first component is limited to a value. Elimination of pairs is only on resolved pairs.

Values are unit, labels, lambdas, and pairs of values; **case**-headers admit values and variables.

Figure 3 defines call-by-value reduction for LDLC. Evaluation contexts are standard. Given all that, the dynamics of LDLC are pleasingly simple. Expression reduction comprises a beta rule for labels, standard beta-value reduction, decomposition of products, and a non-standard rule that transforms a dependent pair into a non-dependent one. Evaluation contexts are standard for left-to-right call-by-value evaluation. For pairs, the dependent form only reduces the first component and the non-dependent form only reduces the second component. Generalized values, terms, and types are closed under value substitution.

3.2 Algorithmic Typing

We start with an algorithmic presentation of typing for LDLC as it is most relevant in the progression to gradual typing.⁵ Algorithmic typing is expressed by a bidirectional system [Pierce and Turner 2000] and comprises several mutually recursive judgments (i.e., algorithms). For clarity, we color output positions in blue and input positions in red.

$\Vdash \Gamma \mathbf{ok}$	context formation (standard);
$\Gamma \Vdash A$	type formation (standard);
$\Gamma \Vdash M \Rightarrow A$	synthesize type A from M ;
$\Gamma \Vdash M \Leftarrow A$	check M against type A ;
$\Gamma \Vdash A \leq B$	check that type A is a subtype of B ;
$\Gamma \Vdash A \Downarrow \Theta(y : B)C$	unfold type A to $\Theta(y : B)C$ accommodating for type conversion
where Θ ranges over Σ and Π indicating the expected type.	

Figure 4 contains the rules for type checking and type synthesis. The checking judgment for type B synthesizes the type A of the term and invokes the subtyping algorithm with inputs A and B . It has two special cases for checking labels and units to avoid loops with the subtyping rule A-SUB-SINGLE .

⁵The appendix contains specification of the algorithmic system along with soundness and completeness results.

Algorithmic type checking and synthesis

$$\begin{array}{c}
\boxed{\Gamma \Vdash M \Leftarrow B} \quad \boxed{\Gamma \Vdash M \Rightarrow A} \\
\\
\text{A-SUB-TYPE} \quad \frac{\Gamma \Vdash M \Rightarrow A \quad \Gamma \Vdash A \leq B}{\Gamma \Vdash M \Leftarrow B} \quad \text{A-VAR} \quad \frac{\Vdash \Gamma, x : A, \Delta \text{ ok}}{\Gamma, x : A, \Delta \Vdash x \Rightarrow A} \quad \text{A-UNIT-I} \quad \frac{\Vdash \Gamma \text{ ok}}{\Gamma \Vdash () \Rightarrow \mathbf{S}\{() : \mathbf{Unit}\}} \\
\\
\text{A-LAB-I} \quad \frac{\Vdash \Gamma \text{ ok}}{\Gamma \Vdash \ell \Rightarrow \mathbf{S}\{\ell : \{\ell\}\}} \quad \text{A-LAB-EL} \quad \frac{\Gamma \Vdash X \Rightarrow \mathbf{S}\{\ell' : L'\} \quad L' \subseteq L \quad \Gamma \Vdash N_{\ell'} \Rightarrow B}{\Gamma \Vdash \mathbf{case} X \{\overline{\ell' : N_{\ell'}^{\ell \in L}}\} \Rightarrow B} \\
\\
\text{A-LAB-EX} \quad \frac{\Gamma \Vdash D \leq L \quad L' = \{\ell \in L \mid \Gamma \Vdash \mathbf{S}\{\ell : \{\ell\}\} \leq D\} \quad (\forall \ell \in L') \Gamma, x : \mathbf{S}\{\ell : D\}, \Delta \Vdash N_{\ell} \Rightarrow B_{\ell}}{\Gamma, x : D, \Delta \Vdash \mathbf{case} x \{\overline{\ell : N_{\ell}^{\ell \in L}}\} \Rightarrow \mathbf{case} x \{\overline{\ell : B_{\ell}^{\ell \in L}}\}} \\
\\
\text{A-LET} \quad \frac{\Gamma \Vdash M \Rightarrow A \quad \Gamma, x : A \Vdash N \Rightarrow B}{\Gamma \Vdash \mathbf{let} x = M \mathbf{in} N \Rightarrow B} \quad x \notin \text{fv}(B) \quad \text{A-PI-I} \quad \frac{\Gamma, x : A \Vdash M \Rightarrow B}{\Gamma \Vdash \lambda(x : A)M \Rightarrow \Pi(x : A)B} \\
\\
\text{A-SIGMA-I} \quad \frac{\Gamma \Vdash M \Leftarrow A \quad \Gamma, x : A \Vdash N \Rightarrow B}{\Gamma \Vdash \langle x : A = M, N \rangle \Rightarrow \Sigma(x : A)B} \quad \text{A-PAIR-I} \quad \frac{\Gamma \Vdash V \Rightarrow A \quad \Gamma \Vdash N \Rightarrow B}{\Gamma \Vdash \langle\langle V, N \rangle\rangle \Rightarrow \Sigma(x : A)B} \quad x \notin \Gamma \\
\\
\text{A-PI-E} \quad \frac{\Gamma \Vdash M \Rightarrow D \quad \Gamma \Vdash D \Downarrow \Pi(x : A)B \quad \Gamma \Vdash N \Leftarrow A \quad \Gamma \Vdash B[N/x]}{\Gamma \Vdash MN \Rightarrow B[N/x]} \\
\\
\text{A-SIGMA-E} \quad \frac{\Gamma \Vdash M \Rightarrow D \quad \Gamma \Vdash D \Downarrow \Sigma(x : A)B \quad \Gamma, x : A, y : B \Vdash N \Rightarrow C}{\Gamma \Vdash \mathbf{let} \langle\langle x, y \rangle\rangle = M \mathbf{in} N \Rightarrow C} \quad x, y \notin \text{fv}(C)
\end{array}$$

Fig. 4. Algorithmic typing in LDLC

The synthesis judgment is designed to compute a minimal type with respect to subtyping. The rules for variables and unit are obvious. The type of a label is the corresponding singleton type (A-LAB-I). For a case expression, we first try rule A-LAB-EL to synthesize a singleton type for the case header X . If we succeed, we synthesize the type from the chosen branch ℓ' , only. Otherwise, the case header must be a variable x and rule A-LAB-EX applies. The type of the variable is checked to be a subtype of L , we synthesize the type of the accessible branches with the type of x specialized to the appropriate singleton label, and collect the types of the branches in a **case** type.

Example 3.1. Rule A-LAB-EL must ignore the non- ℓ' branches. For example, let

$$M = \mathbf{case} x \{T : (), F : 42\} \quad A = \mathbf{case} x \{T : \mathbf{Unit}, F : \mathbf{Int}\}$$

Clearly, M has type A by rule A-LAB-EX. After substitution $M[T/x]$ typechecks by rule A-LAB-EL:

$$M[T/x] = \mathbf{case} T \{T : (), F : 42\} \quad A[T/x] = \mathbf{case} T \{T : \mathbf{Unit}, F : \mathbf{Int}\} \equiv \mathbf{Unit}$$

A slight variation of the example demonstrates why we must ignore the deselected branches.

$$M' = \mathbf{case} x \{T : (), F : 1 + \mathbf{case} x \{T : \text{"foo"}, F : 41\}\} \quad A = \mathbf{case} x \{T : \mathbf{Unit}, F : \mathbf{Int}\}$$

Algorithmic unfolding

$$\Theta \in \{\Pi, \Sigma\} \quad \boxed{\Gamma \Vdash A \Downarrow \Theta(y : B)C}$$

A-U-REFL

$$\Gamma \Vdash \Theta(y : A)B \Downarrow \Theta(y : A)B$$

A-U-CASEL

$$\frac{\Gamma \Vdash X \Rightarrow \mathbf{S}\{\ell' : D\} \quad \ell' \in L \quad \Gamma \Vdash A_{\ell'} \Downarrow \Theta(y : B)C}{\Gamma \Vdash \mathbf{case} X \{\overline{\ell : A_{\ell}}^{\ell \in L}\} \Downarrow \Theta(y : B)C}$$

A-U-CASEX

$$\frac{\begin{array}{c} \Gamma \Vdash D \leq L \quad L' = \{\ell \in L \mid \Gamma \Vdash \mathbf{S}\{\ell : L'\} \leq D\} \\ (\forall \ell \in L') \Gamma, x : \mathbf{S}\{\ell : D\}, \Delta \Vdash A_{\ell} \Downarrow \Theta(y : B_{\ell})C_{\ell} \quad B = \mathbf{case} x \{\overline{\ell : B_{\ell}}^{\ell \in L'}\} \end{array}}{\Gamma, x : D, \Delta \Vdash \mathbf{case} x \{\overline{\ell : A_{\ell}}^{\ell \in L}\} \Downarrow \Theta(y : B) \mathbf{case} x \{\overline{\ell : C_{\ell}}^{\ell \in L'}\}}$$

Algorithmic subtyping

$$\boxed{\Gamma \Vdash A \leq B}$$

A-SUB-UNIT

$$\Gamma \Vdash \mathbf{Unit} \leq \mathbf{Unit}$$

A-SUB-LABEL

$$\frac{L \leq L'}{\Gamma \Vdash L \leq L'}$$

A-SUB-SINGLE

$$\frac{\Gamma \Vdash A \leq B \quad \mathbf{base}(B) \quad \neg \mathbf{S}(B)}{\Gamma \Vdash \mathbf{S}\{V : A\} \leq B}$$

A-SUB-SINGLE-SINGLE

$$\frac{V = W \quad \Gamma \Vdash A \leq B}{\Gamma \Vdash \mathbf{S}\{V : A\} \leq \mathbf{S}\{W : B\}}$$

A-SUB-PI

$$\frac{\Gamma \Vdash A' \leq A \quad \Gamma, x : A' \Vdash B \leq B'}{\Gamma \Vdash \Pi(x : A)B \leq \Pi(x : A')B'}$$

A-SUB-CASE-LL

$$\frac{\Gamma \Vdash X \Rightarrow \mathbf{S}\{\ell' : D\} \quad \ell' \in L \quad \Gamma \Vdash A_{\ell'} \leq B}{\Gamma \Vdash \mathbf{case} X \{\overline{\ell : A_{\ell}}^{\ell \in L}\} \leq B}$$

A-SUB-CASE-XL

$$\frac{\Gamma \Vdash D \leq L \quad L' = \{\ell \in L \mid \Gamma \Vdash \mathbf{S}\{\ell : \{\ell\}\} \leq D\} \quad (\forall \ell \in L') \Gamma, x : \mathbf{S}\{\ell : D\}, \Delta \Vdash A_{\ell} \leq B}{\Gamma, x : D, \Delta \Vdash \mathbf{case} x \{\overline{\ell : A_{\ell}}^{\ell \in L}\} \leq B} \neg \mathbf{S}(D)$$

Fig. 5. Algorithmic unfolding and subtyping in LDLC (excerpt)

The Term M' has type A , but after substitution the F -branch of $M'[T/x]$ does not type check.

$$1 + \mathbf{case} T \{T : \text{"foo"}, F : 41\} \equiv 1 + \text{"foo"}$$

Introduction of a Π -type, A-PI-I, is standard. To eliminate a Π -type, A-PI-E, we first synthesize the type D of the term in function position. We unfold this type expecting a Π -type and extract the argument type A and result type B from the unfolding. It remains to check the argument against type A and to return $B[N/x]$ after checking its well-formedness.

Introduction of a Σ -type, A-SIGMA-I, checks the first component against the type annotation A , synthesizes the type B of the second component, and puts both together. Once the dependency has been resolved by evaluation, A-PAIR-I is the standard rule for pairs. Elimination in A-SIGMA-E proceeds analogous to Π -elimination, and checks that bound variables do not leak into the synthesized type.

One might argue that the label type itself can express singletons already. Singleton types prove their usefulness in connection with the dynamic type in Section 5.

The remaining two judgments, unfolding and subtyping, implement type conversion and subtyping, which are specified in an appendix. Type conversion includes reasoning with singleton types

$G', H' ::= \mathbf{S}\{V : \mathbf{Unit}\} \mid \mathbf{S}\{V : L\} \mid \mathbf{Unit} \mid L$	$\frac{\text{DYN-F} \quad \vdash_C \Gamma \text{ ok}}{\Gamma \vdash_C \star}$	$\frac{\text{BOT-F} \quad \vdash_C \Gamma \text{ ok}}{\Gamma \vdash_C \perp}$	$\frac{\text{TOP-F} \quad \vdash_C \Gamma \text{ ok}}{\Gamma \vdash_C \top}$
$G, H ::= G' \mid \Pi(x : \star) \star \mid \Sigma(x : \star) \star$			

Fig. 6. Ground types and extra formation rules in CCLDLC

to substitute a value V for a variable $z : \mathbf{S}\{V : A\}$ following [Stone and Harper 2006], reducing **case** types analogous to RL-CASE, and eta expansion of **case** types: $\Gamma, z : D, \Delta \vdash A \equiv \mathbf{case} \, z \{ \ell : A[\ell/z] \}^{\ell \in L}$.

Unfolding in Figure 5 is only used to eliminate Π - and Σ -types. It takes two arguments besides the typing environment, the type A to eliminate and the expected type constructor $\Theta \in \{\Pi, \Sigma\}$. If there exist some B and C such that A is convertible to $\Theta(y : B)C$, then unfolding returns such B and C . In other words, unfolding decides whether a given type A is convertible to a Π - or Σ -type.

Algorithmic subtyping, also in Figure 5, deals with two features, singletons and cases, in non-obvious ways. There are two rules for singletons, A-SUB-SINGLE-SINGLE directly compares two singleton types, which requires that the values are equal. A-SUB-SINGLE compares a singleton against a non-singleton type indicated by $\neg \mathbf{S}(B)$. The right side must not be a **case**, indicated by $\neg \mathbf{case}(B)$, because unraveling the case might reveal a singleton type.

The case rules are symmetric. The A-SUB-CASE-L? rules deal with the case where the type of the value confers enough information to decide which branch of the case is taken. To this end, it accepts any singleton type whose value is a member of the set L of expected labels. The A-SUB-CASE-X? rules treat the cases, where there is not enough information about variable $x : D$ to decide the case branch. First, we consider only reachable branches where ℓ is a subtype of D . Second, we narrow the type of x to a singleton type $\mathbf{S}\{\ell : D\}$ when typing the branch for ℓ . This choice enables using one of the A-SUB-CASE-LL/-LR rules when a case on the same variable x is considered again.

PROPOSITION 3.2. *Algorithmic unfolding, subtyping, and typing for LDLC are all decidable.*

PROOF. We can show by mutual rule induction that unfolding and typing give rise to deterministic functions and subtyping is a deterministic predicate. $\square$

The statement of soundness and completeness of algorithmic typing with respect to its specification is available in the extended version.

4 THE CAST CALCULUS FOR LDLC

The programmer-facing gradual language (or *external language*) is obtained from LDLC by introducing the dynamic type and relaxing the type checking algorithm. The semantics of this language is defined by translation to an *internal language*, the cast calculus CCLDLC, which conservatively extends LDLC with the dynamic type and a suitable notion of type casts. As it is important to gain intuition about the execution model, we introduce the cast calculus first and present the external language along with the translation in Section 5.

4.1 Statics of CCLDLC

CCLDLC extends LDLC with the dynamic type, the bottom type, the top type, the cast term $M : A \Rightarrow B$, and the blame term.

$$\begin{aligned}
 A &::= \dots \mid \star \mid \perp \mid \top & M &::= \dots \mid M : A \Rightarrow B \mid \mathbf{blame} \\
 V &::= \dots \mid (V : G \Rightarrow \star) & X &::= x \mid V \mid (x : \star \Rightarrow G) \mid (V : \star \Rightarrow G) \mid \mathbf{blame}
 \end{aligned}$$

A dynamic value $(V : G \Rightarrow \star)$ consists of a type tag G applied to a value V . The type tag takes the form of a *ground type* as described in Figure 6. Ground types are singleton types, base types, as

$$\begin{array}{c}
\text{A-U-BOTPI} \quad \Gamma \Vdash_C \perp \Downarrow \Pi(y : \top) \perp \quad \text{A-U-BOTSIGMA} \quad \Gamma \Vdash_C \perp \Downarrow \Sigma(y : \perp) \perp \quad \text{A-LAB-EL-BOT} \\
\frac{\Gamma \Vdash_C X \Rightarrow \perp}{\Gamma \Vdash_C \text{case } X \{ \ell : \overline{N_\ell^{\ell \in L}} \} \Rightarrow \perp} \\
\\
\text{A-LAB-EX-DYN} \\
\frac{(\forall \ell \in L) \Gamma, x : \text{cast}(\mathbf{S}\{\ell : L\}, L \Rightarrow D), \Delta \Vdash_C N_\ell \Rightarrow B_\ell}{\Gamma, x : D, \Delta \Vdash_C \text{case } (x : D \Rightarrow L) \{ \ell : \overline{N_\ell^{\ell \in L}} \} \Rightarrow \text{case } (x : D \Rightarrow L) \{ \ell : \overline{B_\ell^{\ell \in L}} \} \neg \mathbf{S}(D)}
\end{array}$$

Fig. 7. Additional rules for label elimination and unfolding in CCLDLC

well as Π - and Σ -types where the components are dynamic. Generalized values are extended with blame and a value or variable with a single cast applied to it. This cast is only needed if the value or variable has type dynamic and the result type is a ground type. Type formation in the internal language relies on the rules DYN-F , BOT-F , and TOP-F (Figure 6), and inherits the remaining rules from LDLC: The sole purpose of the bottom type $\perp$ is to provide a type for the **blame** term. The motivation for the top type $\top$ is also purely technical to extend the unfolding function to the $\perp$ type. While the bottom type can appear in a typing environment, the top type is only used in the A-PI-E rule.

It would be sufficient to have declarative typing rules for the cast calculus, but we continue in algorithmic style to make clear that the cast calculus is a conservative extension.⁶ Algorithmic typing for terms of the internal language $\Gamma \Vdash_C M \Rightarrow A$ is defined by the same rules as the LDLC judgment $\Gamma \Vdash M \Rightarrow A$ extended with several typing rules that deal with the new syntactic elements. The typing rule CAST for the cast term places no restriction on source and target type. A reader familiar with the literature might expect a consistency requirement, but such a requirement is not preserved by reduction (cf. Example 4.3).

$$\begin{array}{c}
\text{A-CAST} \quad \frac{\Gamma \Vdash_C M \Rightarrow A' \quad \Gamma \Vdash_C A' \leq A \quad B' = \text{cast}(A', A \Rightarrow B)}{\Gamma \Vdash_C (M : A \Rightarrow B) \Rightarrow B'} \quad \text{A-BLAME} \quad \Gamma \Vdash_C \text{blame} \Rightarrow \perp
\end{array}$$

The CAST rule makes use of the cast function, which makes the operation of the cast more precise. The standard typing of a cast, say, $(x : A \Rightarrow B)$ always returns type B . This interpretation loses precision if x has a singleton type $x : \mathbf{S}\{V : A\}$. This loss is undesirable because some label elimination rules take advantage of singleton types. Hence, type checking attempts to evaluate the cast $(V : A \Rightarrow B)$. If the cast succeeds with value W , the cast function returns the singleton type $\mathbf{S}\{W : B\}$. If the cast fails, it reduces to **blame**, hence the cast returns $\perp$. It returns B in any other case.

$$\text{cast}(A', A \Rightarrow B) = \begin{cases} \mathbf{S}\{W : B\} & \text{if } A' = \mathbf{S}\{V : A\} \text{ and } (V : A \Rightarrow B) \longrightarrow^* W \\ \perp & \text{if } A' = \mathbf{S}\{V : A\} \text{ and } (V : A \Rightarrow B) \longrightarrow^* \text{blame} \\ B & \text{if } A' \leq A \text{ and the first two cases do not apply} \\ \text{undefined} & \text{otherwise} \end{cases}$$

In Figure 7, we extend unfolding to the bottom type: unfolding to a Π -type returns the smallest function type with argument type $\top$ and unfolding to a Σ -type returns the smallest pair type. We also generalize rules from LDLC concerning label elimination. Rule A-LAB-EL obtains a companion rule A-LAB-EL-BOT, which addresses the case where the case header X is a failing cast. Rule A-LAB-EX remains unchanged. There is now an additional possibility for the **case**-header: $(x : D \Rightarrow L)$

⁶See the appendix for the declarative rules.

Algorithmic subtyping

$$\begin{array}{c}
\boxed{\Gamma \Vdash_C A \leq B} \\
\\
\text{A-SUB-BOT} \quad \frac{\Gamma \Vdash_C B}{\Gamma \Vdash_C \perp \leq B} \quad \text{A-SUB-TOP} \quad \frac{\Gamma \Vdash_C A}{\Gamma \Vdash_C A \leq \top} \quad \text{A-SUB-DYN} \quad \Gamma \Vdash_C \star \leq \star \\
\\
\text{A-SUB-CASE-BL} \quad \frac{\Gamma \Vdash X \Rightarrow \perp}{\Gamma \Vdash_C \text{case } X \{ \dots \} \leq B} \\
\\
\text{A-SUB-CASE-CL} \quad \frac{
\begin{array}{c}
L' = \{ \ell \in L \mid \boxed{\Gamma \Vdash_G \{ \ell \} \leq D} \} \\
(\forall \ell \in L') \quad D_\ell = \text{cast}(\mathbf{S}\{\ell : \{ \ell \} \}, \{ \ell \} \Rightarrow D) \quad \Gamma, x : D_\ell, \Delta \Vdash_C A_\ell \leq B
\end{array}
}{\Gamma, x : D, \Delta \Vdash_C \text{case } (x : D \Rightarrow L) \{ \ell : A_\ell^{\ell \in L} \} \leq B} \neg \mathbf{S}(D)
\end{array}$$

Fig. 8. Algorithmic subtyping for CCLDLC (extra rules)

where $x : D$ and $\neg \mathbf{S}(D)$ (If D were a singleton, then the *cast* function enables rule A-LAB-EL). Rule A-LAB-EX-DYN deals with this case. It refines the type of x in each branch under the assumption that the value that will be substituted for x is a label ℓ cast to D .

Example 4.1. Rule A-LAB-EX-DYN gets exercised when a variable $x : \star$ of type dynamic is cast to L in a **case** term. A-LAB-EX-DYN yields the judgment

$$\Gamma, x : \star, \Delta \Vdash_C \text{case } (x : \star \Rightarrow L) \{ \ell : N_\ell^{\ell \in L} \} \Rightarrow \text{case } (x : \star \Rightarrow L) \{ \ell : B_\ell^{\ell \in L} \}$$

In the preservation proof we need to check what happens when substituting a suitable value for the dynamic variable x . There are two choices for “good” and “bad” values. A “good” dynamic value would have the form $V = (\ell : \mathbf{S}\{\ell : \{ \ell \} \} \Rightarrow \star)$. After substitution $[V/x]$, we obtain the typing

$$\Gamma, \Delta \Vdash_C \text{case } (\ell : \mathbf{S}\{\ell : \{ \ell \} \} \Rightarrow \star \Rightarrow L) \{ \ell : N_\ell^{\ell \in L} \} \Rightarrow \text{case } (\ell : \mathbf{S}\{\ell : \{ \ell \} \} \Rightarrow \star \Rightarrow L) \{ \ell : B_\ell^{\ell \in L} \}$$

First, we see that the underlined term still has the form of a generalized value because we substituted a value for x . Second, the typing rule for *cast* reduces the underlined term to ℓ and therefore derives the singleton type $\mathbf{S}\{\ell : L\}$ for it, so that A-LAB-EL is now applicable.

Substituting a “bad” dynamic value like $(() : \mathbf{Unit} \Rightarrow \star)$ for x , we obtain

$$\Gamma, \Delta \Vdash_C \text{case } (() : \mathbf{Unit} \Rightarrow \star \Rightarrow L) \{ \ell : N_\ell^{\ell \in L} \} \Rightarrow \text{case } (() : \mathbf{Unit} \Rightarrow \star \Rightarrow L) \{ \ell : B_\ell^{\ell \in L} \}$$

In this case, $(() : \mathbf{Unit} \Rightarrow \star)$ has type $\mathbf{S}\{(() : \mathbf{Unit} \Rightarrow \star) : \star\}$. Hence, the *cast* $\star \Rightarrow L$ can run on the value and yields **blame**. So the underlined code has type $\perp$ and rule A-LAB-EL-BOT applies:

$$\Gamma, \Delta \Vdash_C \text{case } \text{blame } \{ \ell : N_\ell^{\ell \in L} \} \Rightarrow \perp \quad \longrightarrow \quad \Gamma, \Delta \Vdash_C \text{blame} \Rightarrow \perp$$

Both **case**-headers reduce, the good one to just ℓ of the expected singleton type and the bad one to **blame**. In both cases, the same typing rules apply to the reduced term.

We have to extend conversion and subtyping to deal with generalized values in **case**-types. Figure 8 shows some of the new subtyping rules with respect to LDLC (Figure 5). As before, conversion is implemented as part of subtyping.

There are unsurprising new subtyping rules for the bottom type A-SUB-BOT, the top type A-SUB-TOP, and for the dynamic type A-SUB-DYN. Another new pair of rules is represented by A-SUB-CASE-CL, which scrutinizes a generalized value of the form $(x : D \Rightarrow L)$ where D is not a singleton (in which case a A-SUB-CASE-L? rule applies). It refines its knowledge about the value of x in the branches by calculating an appropriate singleton type $D_\ell = \text{cast}(\mathbf{S}\{\ell : \{ \ell \} \}, \{ \ell \} \Rightarrow D)$, which is potentially dynamic. We omit the symmetric “right” rule. The rule A-SUB-CASE-BL checks a **case** on a generalized

value, which is reducible to **blame** and hence its type is convertible to $\perp$. There is no corresponding right rule because it would just cause the left side to reduce to a $\perp$ type.

Example 4.2. To understand how *cast* works, we calculate the refined type for x in two examples.

- If $D = L'$, then $D_\ell = \mathbf{S}\{\ell : L'\}$ so that the type of x gets refined.
- If $D = \star$, then $D_\ell = \mathbf{S}\{(\ell : \{\ell\} \Rightarrow \star) : \star\}$ so that we get a refined type for $(x : D \Rightarrow L')$.

4.2 Gradual Dynamics

The key point in constructing a dynamics for gradual types is the execution of the casts. Our approach relies on the principle to factor casts between non-base types into functorial casts and ground casts that go forth and back between $\star$ and ground types [Igarashi et al. 2017b; Siek et al. 2015a; Wadler and Findler 2009].

As it turns out, before we can reduce a cast, we have to reduce the types involved. To this end, we characterize the types in head normal form in the internal language with the judgment $\vdash_{nf}$.

$$\vdash_{nf} \perp \quad \vdash_{nf} \star \quad \vdash_{nf} \mathbf{Unit} \quad \vdash_{nf} L \quad \vdash_{nf} \Pi(x : A)B \quad \vdash_{nf} \Sigma(x : A)B \quad \vdash_{nf} \mathbf{S}\{V : A\}$$

We use nA and nB to range over head normal forms of types.

Ground types (Figure 6) represent single type constructors applied to all dynamic arguments. They are the “atoms” of checking equality between two types when executing a cast at run time, as they correspond to type tags used to implement dynamically typed languages.

Case types do not occur in ground types. Ground types arise after factoring a cast (see rules CAST-FACTOR-LEFT and CAST-FACTOR-RIGHT). Factoring happens first, once the cast appears in an evaluation context. At that time the source and target types occurring in the cast are closed normal forms: any case term appearing in the types is reduced before the cast factoring takes place.

The cast calculus has additional values formed by application of a functorial cast to a value. Such values only exist for function types. The cast on a (dependent) pair can always be pushed to the components of the pair value (see reduction CAST-PAIR in Figure 9).

$$V, W ::= \dots \mid V : \Pi(x : A)B \Rightarrow \Pi(x : A')B'$$

To process casts in generalized values, evaluation contexts extend into the header of the **case**.

$$\mathcal{E} ::= \dots \mid \mathbf{case} \mathcal{E} \{ \dots \} \mid (\mathcal{E} : A \Rightarrow B)$$

The very first set of reductions that gets to work on a cast is type reduction $\longrightarrow_T$. Type reduction (in Figure 9) exposes the topmost non-**case** type constructor, which is needed to determine the ground type or applicability of a functorial cast. We first reduce the source type, then the target type of a cast.

$$\begin{array}{c} \text{CAST-REDUCE-LEFT} \\ \frac{A \longrightarrow_T A'}{V : A \Rightarrow B \longrightarrow V : A' \Rightarrow B} \end{array} \qquad \begin{array}{c} \text{CAST-REDUCE-RIGHT} \\ \frac{B \longrightarrow_T B'}{V : nA \Rightarrow B \longrightarrow V : nA \Rightarrow B'} \end{array}$$

Cast factoring requires that the types involved in the cast have been reduced to head normal form. At this point, the ground type is easily obtained.

$$\begin{array}{c} \text{CAST-FACTOR-LEFT} \\ \frac{nA \triangleright G \quad nA \neq G}{(V : nA \Rightarrow \star) \longrightarrow (V : nA \Rightarrow G) : G \Rightarrow \star} \end{array} \qquad \begin{array}{c} \text{CAST-FACTOR-RIGHT} \\ \frac{nB \triangleright H \quad nB \neq H}{(V : \star \Rightarrow nB) \longrightarrow (V : \star \Rightarrow H) : H \Rightarrow nB} \end{array}$$

Figure 9 contains the remaining rules involving casts. The first group defines rules for collapsing casts (if the left ground type is a subtype of the right one) and colliding casts (if there is no subtyping relation). The second group considers casts mediated through the dynamic type. These two rules are well-known from previous work. The third group defines commutation rules for functorial

Type matching

$$\mathbf{Unit} \triangleright \mathbf{Unit} \quad L \triangleright L \quad \Pi(x : A)B \triangleright \Pi(x : *) \star \quad \Sigma(x : A)B \triangleright \Sigma(x : *) \star \quad \mathbf{S}\{V : A\} \triangleright \mathbf{S}\{V : A\}$$

Cast reduction rules

$$\begin{array}{c}
\text{CAST-DYN-DYN} \\
\frac{\cdot \Vdash_C nA \leq \star}{(V : nA \Rightarrow \star) \longrightarrow V} \\
\\
\text{CAST-BOT} \\
V : nA \Rightarrow \perp \longrightarrow \mathbf{blame} \\
\\
\text{CAST-SUB} \\
\frac{\cdot \Vdash_C G' \leq H'}{(V : G' \Rightarrow H') \longrightarrow V} \\
\\
\text{CAST-FAIL} \\
\frac{nA \triangleright G \quad nB \triangleright H \quad \cdot \not\Vdash_C G \leq H}{(V : nA \Rightarrow nB) \longrightarrow \mathbf{blame}} \\
\\
\text{CAST-COLLAPSE} \\
\frac{\cdot \Vdash_C G \leq H}{V : G \Rightarrow \star \Rightarrow H \longrightarrow V} \\
\\
\text{CAST-COLLIDE} \\
\frac{\cdot \not\Vdash_C G \leq H}{V : G \Rightarrow \star \Rightarrow H \longrightarrow \mathbf{blame}} \\
\\
\text{CAST-PAIR} \\
\frac{\langle V, W \rangle : (\Sigma(x : A)B) \Rightarrow (\Sigma(x : A')B') \longrightarrow \langle x : A' = (V : A \Rightarrow A'), (W : B \Rightarrow B') \rangle}{\langle x : A' = (V : A \Rightarrow A'), (W : B \Rightarrow B') \rangle} \\
\\
\text{CAST-FUNCTION} \\
\frac{(V : (\Pi(x : A)B) \Rightarrow (\Pi(x : A')B')) W' \longrightarrow (V (W' : A' \Rightarrow A) : B[(W' : A' \Rightarrow A)/x]) \Rightarrow B'[W'/x]}{(V (W' : A' \Rightarrow A) : B[(W' : A' \Rightarrow A)/x]) \Rightarrow B'[W'/x]}
\end{array}$$

Type reduction

$$\begin{array}{c}
\text{REDUCE-TYPE-EXP} \\
\frac{X \longrightarrow Y}{\mathbf{case} X \{ \ell : A_\ell \} \longrightarrow_T \mathbf{case} Y \{ \ell : A_\ell \}} \\
\\
\text{REDUCE-TYPE-BETA} \\
\mathbf{case} \ell' \{ \ell : A_\ell \} \longrightarrow_T A_{\ell'} \\
\\
\text{REDUCE-TYPE-BLAME} \\
\mathbf{case} \mathbf{blame} \{ \ell : A_\ell \} \longrightarrow_T \perp
\end{array}$$

Fig. 9. Matching, cast reduction, and type reduction in CCLDLC

casts. These rules are adaptations from previous work [Igarashi et al. 2017b; Knowles and Flanagan 2010; Wadler and Findler 2009], but the CAST-PAIR rule appears to be new.

We omit the term reductions that propagate **blame** along the surrounding evaluation context.

Example 4.3. Consistent subtyping (cf. Definition 5.1) for the types in a cast is not preserved by reduction. Assume that $x : \star$ and consider the cast

$$(M : \mathbf{case} (x : \star \Rightarrow \mathbf{Bool}) \{T : \mathbf{Int}, F : \mathbf{Str}\} \Rightarrow \mathbf{Int})$$

This cast may succeed if x turns out to be the value $(T : \mathbf{Bool} \Rightarrow \star)$, so these types are consistent subtypes. But if we are forced to substitute x by $(F : \mathbf{Bool} \Rightarrow \star)$, we obtain the cast

$$(M : \mathbf{case} ((F : \mathbf{Bool} \Rightarrow \star) : \star \Rightarrow \mathbf{Bool}) \{T : \mathbf{Int}, F : \mathbf{Str}\} \Rightarrow \mathbf{Int})$$

where the left type is convertible to **Str**, which is not a subtype of **Int**. After M has reduced to some value V , type reduction eliminates the **case** and exposes the type clash in the cast:

$$(V : \mathbf{Str} \Rightarrow \mathbf{Int}) \longrightarrow \mathbf{blame}$$

4.3 Metatheory

The internal language is a conservative extension of LDLC with type casts. As LDLC is closed under reduction, it is sufficient to demonstrate agreement, type preservation, and progress for CCLDLC.

LEMMA 4.4 (AGREEMENT).

- (1) If $\Gamma \Vdash_C A \leq B$, then $\Gamma \Vdash_C A$ and $\Gamma \Vdash_C B$.
- (2) If $\Gamma \Vdash_C A$, then $\Vdash_C \mathbf{ok}$.
- (3) If $\Gamma \Vdash_C A$ and $\Gamma \Vdash_C A \leq B$, then $\Gamma \Vdash_C B$.
- (4) If $\Gamma \Vdash_C M \Rightarrow A$, then $\Gamma \Vdash_C A$.

THEOREM 4.5 (TYPE PRESERVATION). *If $\cdot \Vdash_C M \Rightarrow A$ and $M \longrightarrow N$, then $\cdot \Vdash_C N \Leftarrow A$.*

THEOREM 4.6 (PROGRESS). *If $\cdot \Vdash_C M \Rightarrow A$, either M is a value, M is **blame**, or $\exists M'$ such that $M \longrightarrow M'$.*

5 GRADUAL LDLC

The design of the gradual version of LDLC involves two steps, which roughly follow the approach of the Gradualizer [Cimini and Siek 2016]. First, we define an external language, GLDLC, targeted at programmers. The typing of this language introduces a dynamic type and “slackens” the typing algorithm such that dynamic types are acceptable at elimination constructs and subtyping is approximated by using the dynamic type as a joker. Thus, typing for GLDLC is defined by an extension of the bidirectional type checking algorithm for LDLC.

Second, we define the semantics of GLDLC by translation into the internal language CCLDLC defined in Section 4. The translation extends the gradual type checking algorithm. It inserts casts precisely where bidirectional typing is in checking mode.

5.1 External Language

After some considerations about specifying GLDLC typing, we proceed immediately to an algorithmic version, which is needed to obtain a deterministic translation to the internal language. The type language and formation rules of GLDLC are exactly the same as for the cast calculus CCLDLC, the term language is taken from LDLC (i.e., there is no cast). The difference lies in the notion of (consistent) subtyping and its incorporation in the type checking algorithm.

We can now give another reason for *not* specifying the typing and subtyping relations for GLDLC. It turns out that there is no satisfactory way to give a rule-based inductive definition for the consistent subtyping relation. The problem is known from previous work, which proposes consistent subtyping relations [Garcia et al. 2016; Igarashi et al. 2017b; Siek and Taha 2007], but it aggravates in the presence of *dependent* subtyping.

Gradual typing requires an approximation to subtyping where any part of a type can be replaced by the dynamic type. That is, we clearly want the (hypothetical) consistent subtyping $\vdash_G A \leq \star$ and $\vdash_G \star \leq B$, for all types A and B . If consistent subtyping were transitive, we would have to bear the undesirable consequence $\vdash_G A \leq B$, i.e., any type is an consistent subtype of any other type. On the other hand, an explicit transitivity rule is needed to enable a sequence of beta/eta reductions and expansions as part of a subtyping. Hence, a declarative specification of consistent subtyping is not viable. The algorithmic subtyping is syntax-driven so as to implement a non-transitive relation.

Similarly, if subsumption had its own free-floating typing rule, then a typing derivation could use two subsumptions to relate any type to any other. Hence, the gradual term typing must be syntax-driven / algorithmic.

To formally connect the algorithms for GLDLC with LDLC, we introduce a non-constructive consistent subtyping relation. To state this definition requires a syntactic precision relation $A \sqsubseteq B$ between gradual types. It states that A is *more precise than* B . Roughly, it means that B can be obtained from A by replacing any number of type-level subterms with $\star$. As GLDLC is dependently-typed, precision has to be defined by mutual induction on types and terms, so there is a companion relation $M \sqsubseteq N$ on GLDLC terms. The base case of the formal definition is $A \sqsubseteq \star$ and all other cases propagate this relation homomorphically.⁷ We lift precision pointwise to type environments.

LDLC types are a proper subset of GLDLC types. We call a GLDLC type/context *static* if it is a LDLC type/context (i.e., the dynamic type $\star$ does not occur in the type/context).

⁷See appendix for a full definition.

$$\begin{array}{c}
\text{A-U-DYN} \qquad \qquad \qquad \text{A-SUB-DYNL} \qquad \qquad \qquad \text{A-SUB-DYNR} \\
\Gamma \Vdash_G \star \Downarrow \Theta(y : \star) \star \qquad \qquad \Gamma \Vdash_G \star \leq B \qquad \qquad \Gamma \Vdash_G A \leq \star \\
\\
\text{A-SUB-PI}^* \\
\frac{\Gamma \Vdash_G A' \leq A \quad \Gamma, x : A' \Vdash_G B \leq B' \quad \text{Just}(L) = \text{scope}(x, B) \cap \text{scope}(x, B') \Rightarrow \Gamma \Vdash_G L \geq A \wedge \Gamma \Vdash_G A' \leq L}{\Gamma \Vdash_G \Pi(x : A)B \leq \Pi(x : A')B'} \\
\\
\text{A-SUB-SIGMA}^* \\
\frac{\Gamma \Vdash_G A \leq A' \quad \Gamma, x : A \Vdash_G B \leq B' \quad \text{Just}(L) = \text{scope}(x, B) \cap \text{scope}(x, B') \Rightarrow \Gamma \Vdash_G L \geq A' \wedge \Gamma \Vdash_G A \leq L}{\Gamma \Vdash_G \Sigma(x : A)B \leq \Sigma(x : A')B'} \\
\\
\text{A-SUB-CASE-XL}^* \\
\frac{\Gamma \Vdash_G D \leq L \quad L' = \{\ell \in L \mid \Gamma \Vdash_G \mathbf{S}\{\ell : \{\ell\}\} \leq D\} \quad (\exists \ell \in L') \Gamma, x : \mathbf{S}\{\ell : D\}, \Delta \Vdash_G A_\ell \leq B}{\Gamma, x : D, \Delta \Vdash_G \mathbf{case } x \{ \ell : A_\ell \}^{\ell \in L} \leq B}
\end{array}$$

Fig. 10. Algorithmic unfolding $\Gamma \Vdash_G A \Downarrow B$ and subtyping $\Gamma \Vdash_G A \leq B$ in GLDLC (abridged)

Definition 5.1 (Consistent conversion and subtyping). Let $\Vdash_G \Gamma \mathbf{ok}$ and $\Gamma \Vdash_G A, B$.

- (1) Define $\Gamma \vdash_G A \equiv B$ (*A and B are consistently convertible under Γ*) if exist static $\Gamma' \sqsubseteq \Gamma$, $A' \sqsubseteq A$, and $B' \sqsubseteq B$ such that $\Gamma' \Vdash A' \leq B'$ and $\Gamma' \Vdash B' \leq A'$.
- (2) Define $\Gamma \vdash_G A \leq B$ (*A is a consistent subtype of B under Γ*) if there exist static $\Gamma' \sqsubseteq \Gamma$, $A' \sqsubseteq A$, and $B' \sqsubseteq B$ such that $\Gamma' \Vdash A' \leq B'$.

This definition has an interesting connection to Abstract Gradual Typing (AGT) [Garcia et al. 2016]. AGT considers a gradual type an abstraction of a set of types. The companion concretization function maps the dynamic type to the set of all types and extends homomorphically over all type constructors. In AGT, two gradual types are consistent iff the intersection of their concretizations is not empty. In the presence of subtyping, AGT calls two types consistent subtypes if there exist elements in their concretizations, which are in the subtype relation. Our definitions extend this idea to include typing contexts to capture dependent subtyping and avoids defining concretizations by direct recourse to the underlying existential.

The definition of algorithmic gradual type synthesis $\Gamma \Vdash_G M \Rightarrow A$ and checking $\Gamma \Vdash_G M \Leftarrow A$ is literally the same as algorithmic typing for the plain language in Figure 4. Graduality is implemented entirely in algorithmic unfolding and consistent subtyping (Figure 10) for GLDLC.

Algorithmic unfolding for GLDLC extends LDLC-unfolding with a single extra rule A-U-DYN that unfolds a dynamic type into a ground type for Π or Σ . This rule serves as the matching relation of the Gradualizer [Cimini and Siek 2016] and the *dom* and *codom* functions of AGT [Garcia et al. 2016]. Recall that unfolding takes the expected type constructor as an argument.

Algorithmic consistent subtyping has one pair of new rules and revises two further pairs compared to the LDLC version. The new pair of rules comprises A-SUB-DYNL and A-SUB-DYNR, that implement $\star \leq B$ and $A \leq \star$, respectively.

The second pair comprises rule A-SUB-CASE-XL* and the symmetric A-SUB-CASE-XR* that address the issue explained in Section 2.5. To identify a *potential* subtype where one of the types involves a **case** on an unknown variable $x : \star$, it is sufficient to find one refinement of x such that subtyping

can be established. This explains the existential quantification in the rules (the unstarred rules for LDLC quantify universally).

The third pair comprises rules A-SUB-PI* and A-SUB-SIGMA* that address a subtle soundness issue. If GLDLC adopted the subtyping rules for Π and Σ types from LDLC without change, then the algorithm would be unsound with respect to consistent subtyping. Consider this example:

$$\cdot \Vdash_G \Pi(x : \mathbf{Unit}) \mathbf{Int} \leq \Pi(x : *) \mathbf{case} \ x \ \{T : \mathbf{Int}, F : \mathbf{Bool}\} \quad (2)$$

The rule A-SUB-PI (from LDLC) generates two subgoals:

$$\cdot \Vdash_G * \leq \mathbf{Unit} \quad (3)$$

$$\cdot, x : * \Vdash_G \mathbf{Int} \leq \mathbf{case} \ x \ \{T : \mathbf{Int}, F : \mathbf{Bool}\} \quad (4)$$

Each is individually acceptable as an consistent subtyping. For (3), we find that

$$\cdot \vdash_G * \leq \mathbf{Unit} \quad (5)$$

because we can choose $\mathbf{Unit} \sqsubseteq *$ on the left side. For (4), we find

$$\cdot, x : * \vdash_G \mathbf{Int} \leq \mathbf{case} \ x \ \{T : \mathbf{Int}, F : \mathbf{Bool}\} \quad (6)$$

because we can choose $x : \mathbf{S}\{T : \mathbf{Bool}\} \sqsubseteq x : *$. These two choices are incompatible so that

$$\cdot \vdash_G \Pi(x : \mathbf{Unit}) \mathbf{Int} \leq \Pi(x : *) \mathbf{case} \ x \ \{T : \mathbf{Int}, F : \mathbf{Bool}\} \quad (7)$$

does *not* hold as we cannot find a suitable static type $A \sqsubseteq *$ such that (7) is a valid subtyping. However, an algorithm using A-SUB-PI would wrongly conclude (2).

The algorithmic subtyping rules for Π and Σ in GLDLC avoid this incompatibility by rejecting types for the bound variable x that are incompatible with the use of x in the body of the type. We observe that a variable can only occur in a type as the expression in a **case**. The metafunction $\text{scope}(x, B)$ traverses the type B in search of x . The function returns *Nothing* if x does not occur in B . Otherwise, it returns $\text{Just}(L)$, where L is the set of labels against which x is certainly tested. The important line in the definition of scope handles the case where x occurs in a **case** expression:

$$\text{scope}(x, \mathbf{case} \ x \ \{\ell : A_\ell\}^{\ell \in L}) = \text{Just}(L) \cap \bigcap_{\ell} \text{scope}(x, A_\ell)$$

Here x is tested against the labels in L . Each branch may contain tests on x that further restrict the domain of x , so we take the intersection of all tests. In this intersection operation, *Nothing* serves as a neutral element. The remaining cases and the definition of $\cap$ may be found in the appendix.

The crucial change in the gradual rule A-SUB-PI* is this extra premise: $\text{Just}(L) = \text{scope}(x, B) \cap \text{scope}(x, B') \Rightarrow \Gamma \Vdash_G L \leq A$. The output $\text{Just}(L)$ from the scope invocations indicates that x occurs in one of the range types and is certainly tested against L . In this case, L must be a subtype of the upper limit A .

In the example (4), we find that $\text{scope}(x, \mathbf{case} \ x \ \{T : \mathbf{Int}, F : \mathbf{Bool}\}) = \text{Just}(\{T, F\})$. As $\{T, F\}$ is not a subtype of $\mathbf{Unit}$, the conclusion (2) is rightfully rejected by the algorithm.

PROPOSITION 5.2. *Algorithmic unfolding, subtyping, and typing for GLDLC are all decidable.*

PROOF. We can show by mutual rule induction that unfolding and typing give rise to deterministic functions and subtyping is a deterministic predicate. $\square$

PROPOSITION 5.3 (SOUNDNESS OF ALGORITHMIC SUBTYPING AND CONVERSION FOR GLDLC). *Let $\Gamma \vdash_G A, B$ and $\Gamma, y : B \vdash_G C$.*

- (1) $\Gamma \Vdash_G A \Downarrow \Theta(y : B)C$ implies $\Gamma \vdash_G A \equiv \Theta(y : B)C$.
- (2) $\Gamma \Vdash_G A \leq B$ implies $\Gamma \vdash_G A \leq B$.

Translating expressions

$$\begin{array}{c}
\boxed{\Gamma \Vdash_G M \Leftarrow A \rightsquigarrow \Gamma' \vdash_C M' : A'} \quad \boxed{\Gamma \Vdash_G M \Rightarrow A \rightsquigarrow \Gamma' \vdash_C M' : A'} \\
\\
\text{T-SUB-TYPE} \\
\frac{\Gamma \Vdash_G M \Rightarrow A \rightsquigarrow \Gamma' \vdash_C M' : A' \quad \Gamma \Vdash_G A \leq B \quad \Gamma \Vdash_G B \rightsquigarrow \Gamma' \vdash_C B'}{\Gamma \Vdash_G M \Leftarrow B \rightsquigarrow \Gamma' \vdash_C (M' : A' \Rightarrow B') : B'} \\
\\
\text{T-LAB-EL} \\
\frac{\Gamma \Vdash_G V \Rightarrow S\{\ell : D\} \rightsquigarrow \Gamma' \vdash_C V' : D' \quad \ell \in L \quad \Gamma \Vdash_G N_\ell \Rightarrow B \rightsquigarrow \Gamma' \vdash_C N'_\ell : B'_\ell}{\Gamma \Vdash_G \text{case } V \{\ell : \bar{N}^{\ell \in L}\} \Rightarrow B \rightsquigarrow \Gamma' \vdash_C \text{case } V' \{\ell : N'_\ell\} : B'_\ell} \\
\\
\text{T-LAB-EX} \\
\frac{\Gamma \Vdash_G D \leq L \quad L' = \{\ell \in L \mid \Gamma \Vdash_G \{\ell\} \leq D\} \quad D_\ell = \text{cast}(S\{\ell : \{\ell\}\}, \{\ell\} \Rightarrow D') \quad (\forall \ell \in L') \Gamma, x : S\{\ell : D\}, \Delta \Vdash_G N_\ell \Rightarrow B_\ell \rightsquigarrow \Gamma', x : D_\ell, \Delta' \vdash_C N'_\ell : B'_\ell}{B = \text{case } x \{\ell : B_\ell^{\ell \in L'}\} \quad B' = \text{case } (x : D' \Rightarrow L) \{\ell : B'_\ell^{\ell \in L'}\} \quad \Gamma, x : D, \Delta \Vdash_G \text{case } x \{\ell : \bar{N}^{\ell \in L}\} \Rightarrow B \rightsquigarrow \Gamma', x : D', \Delta' \vdash_C \text{case } (x : D' \Rightarrow L) \{\ell : N'_\ell^{\ell \in L'}\} : B'} \\
\\
\text{T-PI-E} \\
\frac{\Gamma \Vdash_G M \Rightarrow D \rightsquigarrow \Gamma' \vdash_C M' : D' \quad \Gamma \Vdash_G \Pi(x : A) B \rightsquigarrow \Gamma' \vdash_C \Pi(x : A') B' \quad \Gamma \Vdash_G N \Leftarrow A \rightsquigarrow \Gamma' \vdash_C N' : A' \quad \Gamma \Vdash_G B[N/x]}{\Gamma \Vdash_G MN \Rightarrow B[N/x] \rightsquigarrow \Gamma' \vdash_C (M' : D' \Rightarrow \Pi(x : A') B') N' : B'[N'/x]} \\
\\
\text{T-SIGMA-E} \\
\frac{\Gamma \Vdash_G M \Rightarrow D \rightsquigarrow \Gamma' \vdash_C M' : D' \quad \Gamma \Vdash_G D \Downarrow \Sigma(x : A) B \quad \Gamma \Vdash_G \Sigma(x : A) B \rightsquigarrow \Gamma' \vdash_C \Sigma(x : A') B' \quad \Gamma, x : A, y : B \Vdash_G N \Rightarrow C \rightsquigarrow \Gamma', x : A', y : B' \vdash_C N' : C'}{\Gamma \Vdash_G \text{let } \langle x, y \rangle = M \text{ in } N \Rightarrow C \rightsquigarrow \Gamma' \vdash_C \text{let } \langle x, y \rangle = (M' : D' \Rightarrow \Sigma(x : A') B') \text{ in } N' : C'} \quad x, y \notin \text{fv}(C)
\end{array}$$

Fig. 11. Translation of GLDLC to cast calculus (abridged)

PROOF. By rule induction. $\square$

PROPOSITION 5.4 (COMPLETENESS OF ALGORITHMIC SUBTYPING AND CONVERSION FOR GLDLC).
 Let $\Gamma \vdash_G A, B$ and $\Theta \in \{\Pi, \Sigma\}$.

- (1) $\Gamma \vdash_G A \equiv \Theta(y : B)C$ implies $\Gamma \Vdash_G A \Downarrow \Theta(y : B')C'$ and $\Gamma \vdash_G \Theta(y : B)C \equiv \Theta(y : B')C'$.
- (2) $\Gamma \vdash_G A \leq B$ implies $\Gamma \Vdash_G A \leq B$.

PROOF. Both parts are proved by induction on the combined size of A, B, C and A, B . This complication is caused by our axiomatic definition of consistent convertibility and subtyping. $\square$

5.2 Translation

The translation from the external language to the internal language is given by four translation judgments $\Gamma \Vdash_G M \Rightarrow A \rightsquigarrow \Gamma' \vdash_C M' : A'$ and $\Gamma \Vdash_G M \Leftarrow A \rightsquigarrow \Gamma' \vdash_C M' : A'$, corresponding to type synthesis and type checking, as well as $\Gamma \Vdash_G A \rightsquigarrow \Gamma' \vdash_C A'$ and $\Vdash_G \Gamma \text{ ok} \rightsquigarrow \vdash_C \Gamma' \text{ ok}$ to translate types and typing contexts. The color coding indicates the **inputs** and the **outputs** of the translation as well as highlighting the **judgments of the external language**. The translation extends the algorithmic gradual typing judgments and inherits its soundness and completeness properties.

Figure 11 shows some selected translation rules. Most rules are trivially extended to return copies of the transformed internal counterparts of their subterms. We omit the straightforward definition of the translations of types and contexts. The target part of the translation is intended to be a typing judgment in CCLDLC. For simplicity, the target judgment is **not** algorithmic.

The interesting rules are the consistent subsumption rule T-SUB-TYPE and the unknown label elimination T-LAB-EX. The subsumption rule mediates between type checking against type B (in the conclusion) and type synthesis of type A (in the premise). Soundness of consistent algorithmic

subtyping indicates that there are static types $A_0 \sqsubseteq A$ and $B_0 \sqsubseteq B$ such that $A_0 \leq B_0$ (in LDLC). Hence, the translation inserts a cast from $A' \Rightarrow B'$ in the hope that it may succeed.

The known label elimination T-LAB-EL narrows the branches of a **case** expression to one. It retains the **case** expression in the internal language to simplify proving the gradual guarantee.

The unknown label elimination has a premise indicating that D is an consistent subtype of L and the branches of the **case** are checked with the singleton type $\mathbf{S}\{\ell : D\}$. This singleton type is likely unacceptable in the cast calculus because D might be $\star$. The rule creates an admissible type by applying the *cast* function as explained in Example 4.2.

Although we do not state all rules here, the translation extends into type and environment formation, but not into subtyping where the notion of consistent subtyping is sufficient.

The translation inserts casts at uses of T-SUB-TYPE where the direction flips from checking to synthesis and at unfolding in T-PI-E and T-SIGMA-E. Consistent subtyping requires its argument types to be known, but at this point the components A and B of the Π type (Σ type) are unknown.

The translation is the reason why we need to move to generalized values in **case** headers. When translating an unknown case elimination T-LAB-EX, the scrutinized variable x is replaced by a cast $(x : D' \Rightarrow L)$, which is not a value.

6 METATHEORY OF GRADUAL LDLC

The translation from the external language to the internal language is compatible with typing. Compared to the work of [Cimini and Siek \[2016\]](#) that only had to translate the terms, our result requires translation of terms, types, and typing contexts, thanks to dependent types.

THEOREM 6.1 (TYPING PRESERVATION OF CAST INSERTION). *For all Γ, Γ', M, M' , and A, A' , such that $\Vdash_G \Gamma \text{ ok} \leadsto \vdash_C \Gamma' \text{ ok}$, it holds that:*

- (1) $\vdash_C \Gamma' \text{ ok}$.
- (2) If $\Gamma \Vdash_G A \leadsto \Gamma' \vdash_C A'$, then $\Gamma' \Vdash_C A'$.
- (3) If $\Gamma \Vdash_G M \Rightarrow A \leadsto \Gamma' \vdash_C M' : A'$, then $\Gamma \Vdash_G A \leadsto \Gamma' \vdash_C A'$ and $\Gamma' \Vdash_C M' \Leftarrow A'$.
- (4) If $\Gamma \Vdash_G M \Leftarrow A \leadsto \Gamma' \vdash_C M' : A'$, then $\Gamma \Vdash_G A \leadsto \Gamma' \vdash_C A'$ and $\Gamma' \Vdash_C M' \Leftarrow A'$.

PROOF. By mutual rule induction. □

[Siek et al. \[2015b\]](#) propose a number of correctness criteria for gradual typing, which are expressed by the next few statements culminating in the gradual guarantee (Theorem 6.6 and 6.7).

THEOREM 6.2 (STATIC EMBEDDING). *For all static Γ, M , and A , $\Gamma \Vdash M \Rightarrow A$ iff $\Gamma \Vdash_G M \Rightarrow A$.*

PROOF. Very straightforward mutual rule induction. □

The statement of static simulation requires the notion of term equality up to the presence of identity casts, which we denote with $\Gamma \vdash N' \approx N'' : B$ because it implies contextual equivalence of N' and N'' at type B .

THEOREM 6.3 (STATIC SIMULATION). *Suppose M and A are static and let $\cdot \Vdash_G M \Rightarrow A \leadsto \cdot \vdash_C M' : A'$.*

- (1) M is a value iff M' is a value.
- (2) If $M \longrightarrow N$, then $M' \longrightarrow^* N''$ such that $\cdot \Vdash_G N \Rightarrow B \leadsto \cdot \vdash_C N' : B'$ for some $\cdot \Vdash B \leq A$ and $\cdot \vdash N' \approx N'' : B$.
- (3) If $M' \longrightarrow N'$, then $M \longrightarrow N$ and there exists some N'' such that $\cdot \vdash N' \approx N'' : B$ and $\cdot \Vdash_G N \Rightarrow B \leadsto \cdot \vdash_C N'' : B'$ for some $\cdot \Vdash B \leq A$.

PROOF. Reduction of M is possible because a static GLDLC term is also a term of LDLC, for which reduction is defined and closed. For static terms, the casts introduced by the translation are all trivial identity casts and never lead to blame.

Part (1) is proved by induction on the structure of values.

Part (2) is proved by induction on the LDLC reduction relation.

Part (3) is proved by induction on the CCLDLC reduction relation. $\square$

For the next statement, we consider a language of untyped terms $\underline{M}$ where all type annotations are removed, but with the same dynamics as LDLC, which we denote by $\longrightarrow_U$. We write $\Gamma \vdash_U \underline{M}$ for the judgment that implements scope checking for untyped expressions such that $\cdot \vdash_U \underline{M}$ guarantees that $\underline{M}$ is closed.

THEOREM 6.4 (DYNAMIC EMBEDDING). *There is a translation $[\cdot]$ from the untyped language to CCLDLC such that, for all $\cdot \vdash_U \underline{M}$, its translation has type dynamic $\cdot \vdash_C [\underline{M}] : *$, and:*

- (1) $\underline{M}$ is a value iff $[\underline{M}]$ is a value;
- (2) $\underline{\mathcal{E}}$ is a evaluation context iff $[\underline{\mathcal{E}}]$ is an evaluation context;
- (3) if $\underline{M}$ is stuck, then $[\underline{M}] \longrightarrow \mathcal{E}[\mathbf{blame}]$;
- (4) if $\underline{M} \longrightarrow_U \underline{N}$, then $[\underline{M}] \longrightarrow^2 [\underline{N}]$.

PROOF. By induction on the definition of values, evaluation contexts, and the untyped reduction relation. The superscript ² stands for two reduction steps, the first one for removing the cast and the second one mimicking the successful untyped reduction. $\square$

The monotonicity theorem for the external language needs to be slightly restated with respect to Siek et al. [2015b]. To account for dependency, we also need to be able to vary the typing context.

THEOREM 6.5 (MONOTONICITY WRT PRECISION). *For all $\Gamma \sqsubseteq \Gamma'$, $M \sqsubseteq M'$, A , and B ,*

- (1) $\Vdash_{\Gamma} \Gamma \text{ ok}$ implies $\Vdash_{\Gamma'} \Gamma' \text{ ok}$;
- (2) $\Gamma \Vdash_{\Gamma} A$ implies $\Gamma' \Vdash_{\Gamma} A'$ for some $A \sqsubseteq A'$;
- (3) $\Gamma \Vdash_{\Gamma} A \leq B$ implies $\Gamma' \Vdash_{\Gamma} A' \leq B'$ for all $A \sqsubseteq A'$ and $B \sqsubseteq B'$ such that $\Gamma' \Vdash_{\Gamma} A', B'$;
- (4) $\Gamma \Vdash_{\Gamma} M \Rightarrow A$ implies $\Gamma' \Vdash_{\Gamma} M' \Rightarrow A'$ for some $A \sqsubseteq A'$;
- (5) $\Gamma \Vdash_{\Gamma} M \Leftarrow A$ implies $\Gamma' \Vdash_{\Gamma} M' \Leftarrow A'$ for all $A \sqsubseteq A'$.

PROOF. By rule induction on the translation judgment. $\square$

We proceed in two steps to establish the gradual guarantee. In the first step we show that the translation preserves monotonicity taking the obvious notion of precision for CCLDLC.

THEOREM 6.6 (TRANSLATION PRESERVES MONOTONICITY). *If $M \sqsubseteq N$ and $A \sqsubseteq B$ such that $\cdot \Vdash_{\Gamma} M \Rightarrow A \rightsquigarrow \cdot \vdash_C M' : A'$ and $\cdot \Vdash_{\Gamma} N \Rightarrow B \rightsquigarrow \cdot \vdash_C N' : B'$, then $M' \sqsubseteq N'$ and $A' \sqsubseteq B'$.*

In the second step we show that reduction in the internal language preserves monotonicity. Moreover, if the less precise term reduces to **blame**, then the more precise term also does.

THEOREM 6.7 (REDUCTION PRESERVES MONOTONICITY). *Suppose that $M' \sqsubseteq N'$ do not contain **blame** and $A' \sqsubseteq B'$ such that $\cdot \vdash_C M' : A'$ and $\cdot \vdash_C N' : B'$.*

- M' is a value iff N' is a value.
- $M' \longrightarrow M''$ iff $N' \longrightarrow N''$; if $N'' = \mathcal{F}[\mathbf{blame}]$, then $M'' = \mathcal{E}[\mathbf{blame}]$; if $M'' \neq \mathcal{E}[\mathbf{blame}]$, then $M'' \sqsubseteq N''$.

PROOF. Induction on the definition of values and the reduction relation. $\square$

7 RELATED WORK

Value Dependent Types. Dependent ML [Xi and Pfenning 1999] separates a strongly normalizing index language (simply-typed lambda calculus with equality and primitive operations) from the actual computation. Index terms of type `bool` are used as type guards as well as for type assertions.

F^* [Swamy et al. 2013] is a wide spectrum dependently typed language with linear types and value dependent types. As F^* is mainly geared towards verification it includes a wealth of features, which LDLC does not support, on purpose!

Zombie [Casinghino et al. 2014] combines a general computation language with a specification language via dependent types. This work inspired our treatment for value dependency in Π -types. Presently, LDLC does not distinguish between computation and specification because its computation facilities at the type level are very limited and guaranteed to be terminating.

Idris [Brady 2013] is a dependently typed language with call-by-value evaluation. By choice, Idris accumulates much of best practices in implementing dependently-typed theories. To the best of our knowledge Idris is not restricted to value-dependency, although it appears to be possible to enforce such restrictions [de Muijnck-Hughes et al. 2019].

Nishimura [1998] introduces a calculus for objects where messages (a method name and parameters) are first-class constructs. Each such message is typed as the set of method names that may be invoked by the message, formalized in a second order polymorphic type system.

The lambda calculus fragment of label-dependent session types [Thiemann and Vasconcelos 2020] is very close to LDLC, but differs in several technical details. Their calculus propagates knowledge gained by label elimination using equations of the form $x = \ell$, labels are further generalized to natural numbers with a recursor, Σ -types are not truly dependent, and they have to deal with linearity to safely manage sessions. In contrast, LDLC relies on singleton types for this purpose. Variables of singleton type work like suspended substitutions, but do not mess up inductive hypotheses in proofs. We presently do not support natural numbers, but see no issues in extending LDLC in this direction. Σ -types support true dependency as discussed in Section 3. Linearity is not an issue we have to deal with.

Singleton Types. Aspinall [1994] was the first to study singleton types in the context of subtyping. His system relies on tagged types of the form $S\{M : A\}$, it allows for nested singleton types (e.g., A may also be a singleton), and it targets lambda calculus with full beta reduction.

As nested singleton types complicate the metatheory, Stone [2000] proposes unnested, tagged singleton types and establishes their metatheory with call-by-value evaluation. Later on, Stone and Harper [2006] further simplify the theory by using untagged singleton types restricted to base types (as we do). They show that the resulting system, which includes subtyping, is equally expressive to prior systems with tagged and nested singleton types. LDLC builds on their subtyping rules for singletons and the standard rules for dependent subtyping [Aspinall and Compagnoni 2001].

Gradual Typing. Gradual types are rooted in dynamic typing [Henglein 1994], but have gained new popularity [Siek and Taha 2007, 2006]. Ongoing work explored the connections to contracts and blame [Wadler and Findler 2009] and tried to extend the theory of gradual typing to encompass most type constructions: union and intersection types, polymorphism, effect systems, session types [Ahmed et al. 2011; Castagna and Lanvin 2017; Igarashi et al. 2017b,a; Schwerter et al. 2014; Swamy et al. 2014]. Another line of research explores the design space for implementing gradual typing using threesomes and coercions to express casts [Siek et al. 2015a; Siek and Wadler 2010]. High-level design goals for gradual typing were discussed in the “Refined Criteria for Gradual Typing” [Siek et al. 2015b]. These criteria were found insufficient for some applications (e.g., noninterference in security [Toro et al. 2018]), thus Jacobs et al. [2021] suggest a different criterium based on fully

abstract translation. Further investigation is required to see whether this criterium is applicable to GLDLC, as the setup in that work seems to require recursive types. [New and Licata \[2020\]](#) offer a semantic perspective in terms of their Gradual Type Theory, which argues that gradualization should preserve equational theories. Similar to the modeling of graduality by embedding-projection pairs [[New and Ahmed 2018](#)], their semantic arguments also rely on recursive types / coinduction, which is a non-obvious addition to GLDLC.

Our work is inspired by the Gradualizer [[Cimini and Siek 2016](#)], a methodology to transform a given type system into a corresponding gradual system that fulfills the “Refined Criteria” that we mentioned. Notably, we diverge from that methodology which seems tailored to type systems without subtyping. In particular, the notion of matching proved to be counterproductive: adding matching to our rules would violate the static embedding Theorem 6.2.

Alternatively one might pursue the Abstract Gradual Typing (AGT) approach [[Garcia et al. 2016](#)], which advocates viewing a gradual type as an abstraction of a set of concrete types and justifying the definitions underlying the metatheory and the operational semantics from this abstraction. We provide an axiomatic definition of consistent subtyping that follows the AGT spirit in that it captures the possible subtyping of more precise types. It would be an interesting sanity check of our definitions to develop the abstractions needed for AGT and check whether the relations derived using AGT coincide with our definitions. Moreover, the AGT work provides inductive definitions for the derived relations, which seem out of reach for our axiomatic approach.

Gradual and Dependent. [Ou et al. \[2004\]](#) describe a language that mixes simply-typed and dependently-typed fragments. The compiler automatically inserts coercions when control passes between simple and dependent fragments. Dependent types take the form of refinement types.

[Wadler and Findler \[2009\]](#) consider checking of higher-order contracts with subset types and the dynamic type. However, there is no introduction or elimination rule for subset types. Rather, they are introduced by explicit casts that check the subset predicate at run time.

Hybrid type checking [[Knowles and Flanagan 2010](#)] proposes a theory of dependent functions with arbitrary refinements on base types. The novelty of that work is that typing constraints are first checked with an SMT solver, so that some constraint can be proved or refuted at compile time. The remaining constraints give rise to run-time checks corresponding to coercions/casts.

Gradual certified programming [[Tanter and Tabareau 2015](#)] proposes an avenue to write certified programs starting from programs that defer proof obligations for subset types for decidable predicates to run time. Initially, these programs would be written with placeholders for proof terms which give rise to run-time checks. A run-time check can be elided once a more precise program with a proof term is provided. The authors show that this workflow can be implemented in Coq and is compatible with the standard extraction mechanism.

[Lehmann and Tanter \[2017\]](#) consider a dependently typed language with gradual refinement types. They apply AGT to the problem of gradualizing the logical formulas that characterize refinements (not the underlying simple type structure). That is, a logical formula in this system can be partially unknown, which means that some properties are known at compile time, but further properties are needed, they are checked at run time.

[Dagand et al. \[2018\]](#) consider interoperability between dependently- and simply-typed program fragments. They derive safe coercions from type-theoretic Galois connections and anti-connections. It builds on the refinement approach of [Tanter and Tabareau \[2015\]](#) by encoding an indexed family with a predicate in a refinement.

Several recent papers [[Eremondi et al. 2019](#); [Lennon-Bertrand et al. 2020](#)] consider the application of gradual typing to full scale dependent types, that is, the Calculus of Inductive Constructions (CIC). CIC merges types and terms in a single syntactic category, which forces them to consider

terms like $\star 42$ where there are type-indexed unknown terms besides the unknown (dynamic) type, which cannot happen in GLDLC.

[Lennon-Bertrand et al. \[2020\]](#) tackle gradualization of three variants of the Calculus of Inductive Constructions (Gradual CIC) with varying tradeoffs. This effort is geared towards (partial) verification and certified programming in a full-scale dependently-typed language. They show that it is impossible to define a gradual calculus that is at the same time terminating, suffices graduality, and conservatively extends CIC (the Fire Triangle). This motivates their proposal of three designs that each fulfill two of the properties. Moreover, the mutual dependency between typing, reduction, and precision forces them to define precision semantically (following [\[New and Licata 2020\]](#)), which does not seem to be required in GLDLC.

[Eremondi et al. \[2019\]](#) apply the AGT methodology to a two-stage gradual dependent language GDTL without inductive types. They propose a novel approximate normalization algorithm in the presence of unknown terms and types at compile time (stage one). While approximate normalization is terminating and decidable, evaluation at run time may fail or diverge, as is expected in the presence of dynamic types. The algorithm ensures decidable type checking but loses some static information.

Compared to these works, GLDLC is much more modest. Thanks to stratification we do not have to deal with unknown terms and thanks to our restriction of index calculations to elimination of label types, type-level evaluation always terminates. GLDLC is targeted to indexed programming and interfacing with untyped program fragments (in a scripting language or a Lisp-like language) rather than the development of certified or verified programs. In contrast to related work on GCIC [\[Lennon-Bertrand et al. 2020\]](#) it is *not* possible to replace index terms by $\star$, that is, there is no syntax for an *unknown term* and thus no type of the form $\text{case } \star \{ \dots \}!$. Moreover, our system does not encompass inductive types nor universes. However, there are also similarities, for example, all approaches to gradual dependent types require reduction in the types of casts, they require to add a least type, relax conversion (GLDLC adds the notion of consistent subtyping; there is no subtyping in GCIC or GDTL), all make use of bidirectional typing algorithms. With respect to the Fire Triangle of [Lennon-Bertrand et al. \[2020\]](#), GLDLC drops termination, but fulfills graduality and is conservative with respect to LDLC.

8 CONCLUSIONS

GLDLC is a practically useful system that combines a restricted form value-dependent types and gradual types. Due to these restrictions, GLDLC's syntax is clearly stratified in terms and types so that its design is simpler than other approaches to gradualizing dependent types. On the other hand, our goals are much more modest. Our work is geared towards indexed programming rather than certified programming, which is most clearly indicated by the lack of an equality type in LDLC.

GLDLC comes with a range of technical innovations: full-fledged Σ -types, reduction in casts, a novel interaction between casts and singleton types. The mostly internal use of singleton types is key to the expressiveness of the calculus. It avoids having to support equality and thus simplifies proofs. Subtyping, which usually complicates matters, leads to a system that is surprisingly straightforward to implement and reason about.

We see the main use of GLDLC in seamless interfacing with dynamically-typed languages such as Lisp, Python, or JavaScript. Lisp's symbol type is the blueprint for label values, although this concept is also known as first-class record labels and implemented (in extended version) in OCaml's polymorphic variants. We have demonstrated that GLDLC conveniently models dictionaries and flexible algebraic datatypes safely in a dynamic context.

REFERENCES

- Amal Ahmed, Robert Bruce Findler, Jeremy G. Siek, and Philip Wadler. 2011. Blame For All. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Thomas Ball and Mooly Sagiv (Eds.). ACM, Austin, TX, USA, 201–214.
- David Aspinall. 1994. Subtyping with Singleton Types. In *Computer Science Logic, CSL'94 (LNCS)*, Vol. 933. Springer, Kazimierz, Poland, 1–15.
- David Aspinall and Adriana B. Compagnoni. 2001. Subtyping Dependent Types. *Theoretical Computer Science* 266, 1-2 (2001), 273–309. [https://doi.org/10.1016/S0304-3975\(00\)00175-4](https://doi.org/10.1016/S0304-3975(00)00175-4)
- Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer.
- Edwin Brady. 2013. Idris, A General-Purpose Dependently Typed Programming Language: Design and Implementation. *J. Funct. Program.* 23, 5 (2013), 552–593. <https://doi.org/10.1017/S095679681300018X>
- Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. 2014. Combining Proofs and Programs in a Dependently Typed Language. In *POPL, Suresh Jagannathan and Peter Sewell (Eds.)*. ACM, San Diego, CA, USA, 33–46. <https://doi.org/10.1145/2535838.2535883>
- Giuseppe Castagna and Victor Lanvin. 2017. Gradual Typing with Union and Intersection Types. *PACMPL* 1, ICFP (2017), 41:1–41:28. <https://doi.org/10.1145/3110285>
- Matteo Cimini and Jeremy G. Siek. 2016. The Gradualizer: a Methodology and Algorithm for Generating Gradual Type Systems. In *POPL*. ACM, 443–455.
- Thierry Coquand and Gérard Huet. 1988. The Calculus of Constructions. *Information and Computation* 76, 2/3 (1988), 95–120.
- Pierre-Évariste Dagand, Nicolas Tabareau, and Éric Tanter. 2018. Foundations of Dependent Interoperability. *J. Funct. Program.* 28 (2018), e9. <https://doi.org/10.1017/S0956796818000011>
- Jan de Muijnck-Hughes, Edwin C. Brady, and Wim Vanderbauwhede. 2019. Value-Dependent Session Design in a Dependently Typed Language. In *Proceedings Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2019, Prague, Czech Republic, 7th April 2019 (EPTCS)*, Francisco Martins and Dominic Orchard (Eds.), Vol. 291. 47–59. <https://doi.org/10.4204/EPTCS.291.5>
- Joseph Eremondi, Éric Tanter, and Ronald Garcia. 2019. Approximate Normalization for Gradual Dependent Types. *Proc. ACM Program. Lang.* 3, ICFP (2019), 88:1–88:30. <https://doi.org/10.1145/3341692>
- Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. Abstracting Gradual Typing. In *POPL*. ACM, 429–442.
- Fritz Henglein. 1994. Dynamic Typing: Syntax and Proof Theory. *Science of Computer Programming* 22 (1994), 197–230.
- Atsushi Igarashi, Peter Thiemann, Vasco T. Vasconcelos, and Philip Wadler. 2017b. Gradual Session Types. *Proc. ACM Program. Lang.* 1, ICFP, Article 38 (Sept. 2017), 28 pages. <https://doi.org/10.1145/3110282>
- Yuu Igarashi, Taro Sekiyama, and Atsushi Igarashi. 2017a. On Polymorphic Gradual Typing. *PACMPL* 1, ICFP (2017), 40:1–40:29. <https://doi.org/10.1145/3110284>
- Koen Jacobs, Amin Timany, and Dominique Devriese. 2021. Fully Abstract from Static to Gradual. *Proc. ACM Program. Lang.* 5, POPL (2021), 1–30. <https://doi.org/10.1145/3434288>
- Kenneth L. Knowles and Cormac Flanagan. 2010. Hybrid type checking. *ACM Trans. Program. Lang. Syst.* 32, 2 (2010), 6:1–6:34.
- Fabian Krause, Weili Fu, and Peter Thiemann. 2021. Artifact for Label Dependent Lambda Calculus and Gradual Typing. <https://doi.org/10.5281/zenodo.5497628>
- Nico Lehmann and Éric Tanter. 2017. Gradual Refinement Types. In *POPL*. ACM, Paris, France, 775–788.
- Meven Lennon-Bertrand, Kenji Maillard, Nicolas Tabareau, and Éric Tanter. 2020. Gradualizing the Calculus of Inductive Constructions. *CoRR* abs/2011.10618 (2020). arXiv:2011.10618 <https://arxiv.org/abs/2011.10618>
- Per Martin-Löf. 1984. *Intuitionistic Type Theory*. Bibliopolis, Napoli.
- Max S. New and Amal Ahmed. 2018. Graduality from Embedding-Projection Pairs. *Proc. ACM Program. Lang.* 2, ICFP (2018), 73:1–73:30. <https://doi.org/10.1145/3236768>
- Max S. New and Daniel R. Licata. 2020. Call-by-name Gradual Type Theory. *Logical Methods in Computer Science* Volume 16, Issue 1 (Jan. 2020). [https://doi.org/10.23638/LMCS-16\(1:7\)2020](https://doi.org/10.23638/LMCS-16(1:7)2020)
- Susumu Nishimura. 1998. Static Typing for Dynamic Messages. In *Proc. 25th ACM Symp. POPL*, Luca Cardelli (Ed.). ACM Press, San Diego, CA, USA, 266–278. <https://doi.org/10.1145/268946.268968>
- Ulf Norell. 2008. Dependently Typed Programming in Agda. In *Advanced Functional Programming (LNCS)*, Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra (Eds.), Vol. 5832. Springer, Heijten, The Netherlands, 230–266.
- Xinming Ou, Gang Tan, Yitzhak Mandelbaum, and David Walker. 2004. Dynamic Typing with Dependent Types. In *IFIP TCS*, Jean-Jacques Lévy, Ernst W. Mayr, and John C. Mitchell (Eds.). Kluwer, 437–450.
- Benjamin C. Pierce and David N. Turner. 2000. Local Type Inference. *ACM TOPLAS* 22, 1 (2000), 1–44. <https://doi.org/10.1145/345099.345100>

- Felipe Bañados Schwerter, Ronald Garcia, and Éric Tanter. 2014. A theory of gradual effect systems. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, Gothenburg, Sweden, 283–295. <https://doi.org/10.1145/2628136.2628149>
- Jeremy Siek and Walid Taha. 2007. Gradual Typing for Objects. In *21st ECOOP (LNCS)*, Erik Ernst (Ed.), Vol. 4609. Springer, Berlin, Germany, 2–27. https://doi.org/10.1007/978-3-540-73589-2_2
- Jeremy Siek, Peter Thiemann, and Philip Wadler. 2015a. Blame and Coercion: Together Again for the First Time. In *PLDI*, Steve Blackburn (Ed.). ACM, Portland, OR, USA, 425–435. <https://doi.org/10.1145/2737924.2737968>
- Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*.
- Jeremy G. Siek and Manish Vachharajani. 2008. Gradual Typing with Unification-Based Inference. In *Proceedings of the 2008 Symposium on Dynamic Languages, DLS 2008, July 8, 2008, Paphos, Cyprus*, Johan Brichau (Ed.). ACM, 7. <https://doi.org/10.1145/1408681.1408688>
- Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015b. Refined Criteria for Gradual Typing. In *SNAPL (LIPIcs)*, Vol. 32. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 274–293.
- Jeremy G. Siek and Philip Wadler. 2010. Threesomes, With and Without Blame. In *Proc. 37th ACM Symp. POPL*. ACM Press, Madrid, Spain, 365–376. <https://doi.org/10.1145/1706299.1706342>
- Vilhelm Sjöberg, Chris Casinghino, Ki Yung Ahn, Nathan Collins, Harley D. Eades III, Peng Fu, Garrin Kimmell, Tim Sheard, Aaron Stump, and Stephanie Weirich. 2012. Irrelevance, Heterogeneous Equality, and Call-by-value Dependent Type Systems. In *Proceedings Fourth Workshop on Mathematically Structured Functional Programming (EPTCS)*, James Chapman and Paul Blain Levy (Eds.), Vol. 76. Tallinn, Estonia, 112–162. <https://doi.org/10.4204/EPTCS.76.9>
- Christopher Stone. 2000. *Singleton Kinds and Singleton Types*. Ph.D. Dissertation. School of Computer Science, CMU.
- Christopher A. Stone and Robert Harper. 2006. Extensional Equivalence and Singleton Types. *ACM Trans. Comput. Log.* 7, 4 (2006), 676–722. <https://doi.org/10.1145/1183278.1183281>
- Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. 2013. Secure Distributed Programming With Value-Dependent Types. *J. Funct. Program.* 23, 4 (2013), 402–451. <https://doi.org/10.1017/S0956796813000142>
- Nikhil Swamy, Cédric Fournet, Aseem Rastogi, Karthikeyan Bhargavan, Juan Chen, Pierre-Yves Strub, and Gavin M. Bierman. 2014. Gradual Typing Embedded Securely in JavaScript. In *POPL*. ACM, 425–438.
- Éric Tanter and Nicolas Tabareau. 2015. Gradual Certified Programming in Coq. In *Proceedings of the 11th Symposium on Dynamic Languages, DLS 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Manuel Serrano (Ed.). ACM, 26–40. <https://doi.org/10.1145/2816707.2816710>
- Peter Thiemann and Vasco T. Vasconcelos. 2020. Label-Dependent Session Types. *Proc. ACM Program. Lang.* 4, POPL (2020), 67:1–67:29. <https://doi.org/10.1145/3371135>
- Matías Toro, Ronald Garcia, and Éric Tanter. 2018. Type-Driven Gradual Security with References. *ACM Trans. Program. Lang. Syst.* 40, 4 (2018), 16:1–16:55. <https://doi.org/10.1145/3229061>
- typescript handbook 2021. TypeScript Handbook, Advanced Types. <https://www.typescriptlang.org/docs/handbook/advanced-types.html>.
- Philip Wadler and Robert Bruce Findler. 2009. Well-Typed Programs Can’t Be Blamed. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009 (Lecture Notes in Computer Science)*, Giuseppe Castagna (Ed.), Vol. 5502. Springer, York, UK, 1–16. <https://doi.org/10.1007/978-3-642-00590-9>
- Stephanie Weirich, Pritam Choudhury, Antoine Voizard, and Richard A. Eisenberg. 2019. A Role for Dependent Types in Haskell. *Proc. ACM Program. Lang.* 3, ICFP (2019), 101:1–101:29. <https://doi.org/10.1145/3341705>
- Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *Proc. 26th ACM Symp. POPL*, Alexander Aiken (Ed.). ACM Press, San Antonio, Texas, USA, 214–227.