Contextual Typing

XU XUE, The University of Hong Kong, China
BRUNO C. D. S. OLIVEIRA, The University of Hong Kong, China

Bidirectional typing is a simple, lightweight approach to type inference that propagates known type information during typing, and can scale up to many different type systems and features. It typically only requires a reasonable amount of annotations and eliminates the need for many obvious annotations. Nonetheless the power of inference is still limited, and complications arise in the presence of more complex features.

In this paper we present a generalization of bidirectional typing called *contextual typing*. In contextual typing not only known type information is propagated during typing, but also other known information about the *surrounding context* of a term. This information can be of various forms, such as other terms or record labels. Due to this richer notion of contextual information, less annotations are needed, while the approach remains lightweight and scalable. For type systems with subtyping, contextual typing subsumption is also more expressive than subsumption with bidirectional typing, since partially known contextual information can be exploited. To aid specifying type systems with contextual typing, we introduce **Q**uantitative **T**ype **A**ssignment **S**ystem**s** (QTASs). A QTAS quantifies the amount of type information that a term needs in order to type check using counters. Thus, a counter in a QTAS generalizes modes in traditional bidirectional typing, which can only model an all (checking mode) or nothing (inference mode) approach. QTASs enable precise guidelines for annotatability of contextual type systems formalized as a theorem. We illustrate contextual typing first on a simply typed lambda calculus, and then on a richer calculus with subtyping, intersection types, records and overloading. All the metatheory is formalized in the Agda theorem prover.

CCS Concepts: • **Theory of computation** → **Type theory**.

Additional Key Words and Phrases: Bidirectional Typing, Type Inference

ACM Reference Format:
Xu Xue and Bruno C. d. S. Oliveira. 2024. Contextual Typing. *Proc. ACM Program. Lang.* 8, ICFP, Article 266 (August 2024), 29 pages. https://doi.org/10.1145/3674655

1 Introduction

Bidirectional typing is a technique for the design and implementation of type systems. It has been popularized by Pierce and Turner [2000], and surveyed by Dunfield and Krishnaswami [2022]. The key idea is to have two modes for typing: *inference* and *checking*, which describe local type information propagation. The type inferred from a term can be further used to check other terms. In other words, the type information can be propagated from one term to its neighboring terms. Bidirectional typing has proved useful for the design of complex type systems with a variety of features, including subtyping [Davies and Pfenning 2000; Dunfield and Pfenning 2004; Pierce and Turner 2000], polymorphism [Dunfield and Krishnaswami 2013; Zhao et al. 2019] and dependent types [Asperti et al. 2012; Coquand 1996; Löh et al. 2010; Norell 2007; Xi and Pfenning 1999].

Bidirectional typing eliminates the need for many obvious annotations, but the power of inference is still limited. Standard bidirectional typing is an all-or-nothing approach: either all type

Authors' Contact Information: Xu Xue, The University of Hong Kong, Hong Kong, China, xxue@cs.hku.hk; Bruno C. d. S. Oliveira, The University of Hong Kong, Hong Kong, China, bruno@cs.hku.hk.

ACM 2475-1421/2024/8-ART266
https://doi.org/10.1145/3674655

information is known and we can type check a term; or no type information is known and we can infer the type of a term. However, several researchers [Bierman et al. 2014; Norell 2007; Odersky et al. 2001; Polikarpova et al. 2016; Pottier and Régis-Gianas 2006; Xie and Oliveira 2018] observed that *partially known* type information is helpful to improve inference. In some of those works, a notion that naturally arises is some form of contextual type information that aids type inference.

In this paper we present a generalization of bidirectional typing called *contextual typing*. Contextual typing builds on the idea that partially known type information is helpful for inference, and further extends it. In contextual typing not only known type information is propagated during typing, but also other (partially) known information about the *surrounding context* of a term. This information can be of various forms, such as other terms or record labels. For instance, for terms representing functions, it could be the terms (arguments) that the function is being applied to. Conversely, for records, it could be the information about labels that are being projected. Due to this richer notion of contextual information, fewer annotations are needed, while the approach remains lightweight and scalable. Contextual typing retains the lightweightness of bidirectional typing, by simply propagating some information from some terms into some other neighboring terms. No advanced mechanisms, such as unification variables, are needed for contextual typing.

For type systems with subtyping, contextual typing subsumption is also more expressive than bidirectional typing subsumption. Unlike the standard bidirectional subsumption rule, which requires full type information for the supertype of a term, contextual typing subsumption can be used when only partial contextual information is known. Contextual subsumption is helpful to eliminate some complications that arise when applying bidirectional typing to type systems with subtyping. In particular, in various type systems with subtyping – including features like intersection types and/or union types [Davies and Pfenning 2000; Huang et al. 2021; Rioux et al. 2023], records [Xue et al. 2022] or polymorphism [Dunfield and Krishnaswami 2013; Zhao et al. 2019] – specialized auxiliary relations are needed in other parts of the typing relation to compensate for the weak form of bidirectional subsumption. Because contextual typing has a more powerful form of subsumption, it can avoid many such auxiliary relations. Thus we can have a typing relation where all uses of subtyping are delegated to the subsumption rule, and dealt with in a single place.

To specify type systems with contextual typing we introduce **Q**uantitative **T**ype **A**ssignment **S**yste**m**s (QTASs). QTASs serve as an intermediate step in between traditional Type Assignment Systems (TASs) and algorithmic formulations. Unlike type assignment systems, QTASs precisely specify where type annotations are needed. Moreover, they *quantify* (and in some cases qualify) the amount of type information that a term needs in order to type check via counters. Thus, the notion of counters in a QTAS *generalizes* modes in traditional bidirectional typing from an all (checking mode) or nothing (inference mode) approach, to a notion that quantifies the amount of needed type information. With a QTAS it is possible to have precise guidelines for the annotatability of type systems with contextual typing formalized as a theorem.

We illustrate the key ideas of contextual typing and propagation of partial contextual information contextual typing first on a simplified setting based on the simply typed lambda calculus. In this calculus, contextual information consists of known type information, and known arguments to functions. Then we present a richer calculus with subtyping, intersection types, records and a simple form of overloading. This calculus illustrates a more powerful form of subsumption provided by contextual typing, and shows that we can have other forms of contextual information. In particular, we use labels of records as contextual information to aid with the type inference of record projections. For both calculi we prove several results, including soundness and completeness between a QTAS and its algorithmic formulation, decidability of typing and/or subtyping, and annotatability theorems. All the metatheory is formalized in the Agda theorem prover.

$\boxed{\Gamma \vdash e \Leftrightarrow A}$ (Under environment Γ, e can infer ($\Rightarrow$) the type A or check ($\Leftarrow$) against the type A.)

$$\frac{}{\Gamma \vdash i \Leftarrow \texttt{Int}}\ \texttt{Lit} \qquad \frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A}\ \texttt{Var} \qquad \frac{\Gamma \vdash e \Rightarrow A \qquad A = B}{\Gamma \vdash e \Leftarrow B}\ \texttt{Sub} \qquad \frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash (e : A) \Rightarrow A}\ \texttt{Anno}$$

$$\frac{\Gamma, x : A \vdash e \Leftarrow B}{\Gamma \vdash (\lambda x.\ e) \Leftarrow A \to B}\ \texttt{Lam} \qquad \frac{\Gamma \vdash e_1 \Rightarrow A \to B \qquad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\ e_2 \Rightarrow B}\ \texttt{App}$$

Fig. 1. Bidirectional typing of STLC. Note that $\Leftrightarrow$ is a metavariable defined as: $\Leftrightarrow ::= \Leftarrow \mid \Rightarrow$.

In summary, the contributions of this paper are:

- **Contextual typing:** A generalization of bidirectional typing, which retains its lightweightness, and enables programs to type check with fewer annotations.
- **Quantitative type assignment systems.** A variant of type assignment systems, which specifies the amount of information needed to type check terms, and enables precise annotatability guidelines to be formalized as theorems.
- **A teleportation-based approach for algorithmic typing.** To implement a QTAS, we employ a typing relation that, in addition to a type environment, is also parametrized by a *surrounding context*. The surrounding context tracks known contextual type information, and is used to aid with the propagation (or teleportation) of information across the AST.
- **The metatheory for contextual typing.** All the calculi and proofs in this paper are formalized in Agda and they are available in the companion artifact [Xue and Oliveira 2024].
- **A contextual intersection type system with subtyping, records and overloading,** illustrating how contextual typing scales up to subtyping and some common features.

2 Bidirectional Typing: Some Variants and Limitations

In this section we first introduce a standard formulation of *bidirectional typing* [Dunfield and Krishnaswami 2022; Pierce and Turner 2000], and an alternative bidirectional approach called *let arguments go first* [Xie and Oliveira 2018]. We then discuss limitations of both forms of bidirectional typing, providing us with the motivation for the contextual typing approach.

2.1 Bidirectional Typing

Bidirectional STLC. To illustrate bidirectional typing, we use the variant of the Simply Typed Lambda Calculus (STLC) by Dunfield and Krishnaswami as an example in Fig. 1. The original calculus uses unit and unit types, which we replace by integers and integer types. This calculus is designed based on a design recipe by Dunfield and Pfenning [2004], called the *Pfenning recipe*. This recipe gives a guideline on how to bidirectionalize type assignment systems. Intuitively, we first find the principal judgment and follow the rules: introduction forms check types, and elimination forms infer types. The `Lit` and `Var` rules are two base rules: i can check against the `Int` type and variable x can infer the type A by looking up the typing environment Γ. The checking rule for integers strictly follows the Pfenning recipe. The subsumption rule `Sub` switches from checking to inference mode by comparing the type equality between the inferred and the checked types. The annotation rule `Anno` infers the type A from the annotation and uses it to check the term e. In the `Lam` rule, the lambda term $\lambda x.\ e$ checks against a function type $A \to B$, and the output type B will be used to check the body e in an extended environment. The application rule `App` is the most interesting one. In order to infer the type of the application, we first infer e_1's type and obtain the function type, then use its input type A to check the argument e_2 and use its output type B as the inference result of the application. For example, we can infer the type of $((\lambda f.\ f\ 1) : (I \to I) \to I)\ (\lambda x.\ x)$, where

the annotated lambda expecting a function f is applied to a raw lambda (I stands for `Int`).

$$\dfrac{\dfrac{\cdots}{\Gamma \vdash (\lambda f.\ f\ 1) : (I \to I) \to I \Rightarrow (I \to I) \to I}\ \textsf{Anno} \qquad \dfrac{\cdots}{\Gamma \vdash \lambda x.\ x \Leftarrow I \to I}\ \textsf{Lam}}{\Gamma \vdash ((\lambda f.\ f\ 1) : (I \to I) \to I)\ (\lambda x.\ x) \Rightarrow I}\ \textsf{App}$$

The derivation infers a function type $(I \to I) \to I$. The input type $I \to I$ is used to check the argument $\lambda x.\ x$, and the output type I is used as the inference result of the application.

Mode-correctness. The possibility of using two modes for typing brings up the question of when to use one mode or the other in both the premises and the conclusions. The choices in Fig. 1 are not unique, and there are several alternative ways to use modes, and still obtain a set of rules that can be directly implemented as an algorithm. However, there are some choices for the usage of the modes that are *non-algorithmic*. It is possible to design bidirectional rules where types need to be guessed. Dunfield and Krishnaswami suggest the notion of mode-correctness as a criterion to ensure the possibility of a bidirectional rule being implementable. Intuitively, we must avoid guessing a type when we design bidirectional typing rules. When we check the term in the premise by some type, the type must come from other synthesized terms or the checking type in the conclusion. When we infer the type of the conclusion, the type should come from synthesized results of premises. Dunfield and Krishnaswami identified four different mode-correct application rules. Among those rules, `App1` is the default choice for most bidirectional type systems.

$$\dfrac{\Gamma \vdash e_1 \Rightarrow A \to B \qquad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\ e_2 \Rightarrow B}\ \textsf{App1} \qquad \dfrac{\Gamma \vdash e_2 \Rightarrow A \qquad \Gamma \vdash e_1 \Leftarrow A \to B}{\Gamma \vdash e_1\ e_2 \Leftarrow B}\ \textsf{App2}$$

$$\dfrac{\Gamma \vdash e_1 \Rightarrow A \to B \qquad \Gamma \vdash e_2 \Rightarrow A}{\Gamma \vdash e_1\ e_2 \Rightarrow B}\ \textsf{App3} \qquad \dfrac{\Gamma \vdash e_1 \Rightarrow A \to B \qquad \Gamma \vdash e_2 \Leftarrow A}{\Gamma \vdash e_1\ e_2 \Leftarrow B}\ \textsf{App4}$$

Note that `App3` and `App4` can be subsumed by `App1` with the subsumption rule, but not `App2`, which does add expressive power to the type system. With `App2`, we can annotate the application and utilize the inference result of arguments to further check the function. For example, $((\lambda x.\ x)\ z) : \texttt{Int}$ cannot infer a type for a system with only `App1`, but type checks with `App2` included.

$$\dfrac{\dfrac{\dfrac{\cdots}{\Gamma, z : \texttt{Int} \vdash \lambda x.\ x \Leftarrow \texttt{Int} \to \texttt{Int}}\ \textsf{Lam} \qquad \dfrac{(z : \texttt{Int}) \in \Gamma, z : \texttt{Int}}{\Gamma, z : \texttt{Int} \vdash z \Rightarrow \texttt{Int}}\ \textsf{Var}}{\Gamma, z : \texttt{Int} \vdash (\lambda x.\ x)\ z \Leftarrow \texttt{Int}}\ \textsf{App2}}{\Gamma, z : \texttt{Int} \vdash ((\lambda x.\ x)\ z) : \texttt{Int} \Rightarrow \texttt{Int}}\ \textsf{Anno}$$

Backtracking. Both `App1` and `App2` can type check terms that cannot be type checked with the other rule alone. Thus, one may wonder if it is possible to have both rules in the same type system. This is possible, and it can be implemented. However, simply adding the `App2` rule comes at a cost: we may need to backtrack when type checking an application. Since the two application rules do not subsume each other we may need to try both rules for some applications in order to make type checking work. For instance, when type checking expressions such as $(((\lambda f.\ f\ 1) : (I \to I) \to I)\ (\lambda x.\ x)) : I$, we have to check the application with the type `Int`. In an implementation, we would typically give less priority to the subsumption rule, in order to try other checking rules first. So, the natural choice would be to try to use `App2` first in this case. In this case `App2` fails, then we try the subsumption rule (`Sub`), and `App1` succeeds. Since applications are pervasive in programs, backtracking is problematic as it can introduce significant slowdowns in the type checker. Thus, many implementations would avoid having the two rules and would typically prefer `App1`.

$\boxed{\Gamma \mid \Psi \vdash e \Rightarrow A}$ (Under environment Γ and application stack Ψ, e infers the type A)

$$\frac{}{\Gamma \vdash i \Rightarrow \texttt{Int}}\ \texttt{Lit} \qquad \frac{(x : A) \in \Gamma}{\Gamma \vdash x \Rightarrow A}\ \texttt{Var} \qquad \frac{\Gamma, x : A \mid \Psi \vdash e \Rightarrow B}{\Gamma \mid \Psi, A \vdash \lambda x.\ e \Rightarrow A \to B}\ \texttt{Lam}$$

$$\frac{\Gamma \vdash e_2 \Rightarrow A \qquad \Gamma \mid \Psi, A \vdash e_1 \Rightarrow A \to B}{\Gamma \mid \Psi \vdash e_1\ e_2 \Rightarrow B}\ \texttt{AppS}$$

Fig. 2. Let arguments go first, and the application mode for the STLC.

2.2 Let Arguments Go First

The two previous application rules are useful. However, there are programs that appear to have enough contextual information, but do not type check by any of the mode-correct application rules in bidirectional typing. For instance, to infer the type of $(\lambda x.\ \lambda y.\ x + y)\ 1\ 2$, we can neither pick App1 nor App2. App1 cannot be applied because $\lambda x.\ \lambda y.\ x + y$ is a raw lambda that infers nothing. App2 cannot be applied because the application is not annotated thus we know nothing about the output type. The approach with bidirectional typing is all-or-nothing and forbids partial type information to be used (i.e. information about the inputs of functions only) to assist inference. Nonetheless, type information about arguments can be used to infer the type of raw lambdas. This problem has been observed by Xie and Oliveira [2018], and an alternative formulation of bidirectional typing has been proposed. Instead of combining checking and inference modes, there is a combination of *application* and inference modes. We present the STLC version of this approach in Fig. 2.

The key idea is to introduce an extra environment, called the *application stack* Ψ, which stores *contextual* type information about the types of the arguments. The inference mode then becomes a special case of the application mode when the stack is empty. In other words, $\Gamma \vdash e \Rightarrow A$ is syntactic sugar that stands for $\Gamma \mid . \vdash e \Rightarrow A$. We omit the stack in rule Lit and Var to emphasize the use of the inference mode in those rules. AppS and Lam are the two rules that interact with the application stack. Rule AppS first infers the type A from argument e_2, and pushes the type A into the application stack of e_1. With the help of an extended stack, e_1 infers the function type $A \to B$, and the output type is used as the inference result of the application term. In rule Lam, we pop one type from the stack and use it as the type of bound variable, and then we further infer the body of lambda. With the application mode, $(\lambda x.\ \lambda y.\ x + y)\ 1\ 2$ is typeable as shown in the derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\cdots}{\Gamma, x : I, y : I \mid . \vdash x + y \Rightarrow I}\ \texttt{AppS}}{\Gamma, x : I \mid I \vdash \lambda y.\ x + y \Rightarrow I \to I}\ \texttt{Lam}}{\Gamma \mid I, I \vdash \lambda x.\ \lambda y.\ x + y \Rightarrow I \to I \to I}\ \texttt{Lam} \qquad \dfrac{}{\Gamma \vdash 1 \Rightarrow I}\ \texttt{Lit}}{\Gamma \mid I \vdash (\lambda x.\ \lambda y.\ x + y)\ 1 \Rightarrow I \to I}\ \texttt{AppS} \qquad \dfrac{}{\Gamma \vdash 2 \Rightarrow I}\ \texttt{Lit}}{\Gamma \mid . \vdash (\lambda x.\ \lambda y.\ x + y)\ 1\ 2 \Rightarrow I}\ \texttt{AppS}$$

Let expressions as sugar. Standard bidirectional typing requires extra annotations on lambdas or applications to type $(\lambda x.\ e_2)\ e_1$. One virtue of the application mode is that let-expressions can be simply encoded as syntactic sugar, without adding new constructs or new typing rules:

$$\texttt{let}\ x = e_1\ \texttt{in}\ e_2 \triangleq (\lambda x.\ e_2)\ e_1$$

Limitations. Unfortunately, the AppS rule excludes cases that standard bidirectional typing can accept. For example, the term $((\lambda f.\ f\ 1) : (I \to I) \to I)\ (\lambda x.\ x)$, cannot be typed with the rules in Fig 2. More interestingly, even though AppS appears to be related to App2, since both rules require the inference of the argument, AppS cannot subsume App2 either. For example, AppS cannot type check the expression $((\lambda x.\ \lambda y.\ y)\ 1) : I \to I$ because it does not deal with annotated terms. Thus, it

cannot propagate the outer annotation to the subterms of the application. In essence, the power of the checking mode is missing. As acknowledged by Xie and Oliveira, the application mode is stronger than the inference mode, but weaker than the checking mode.

To combine the power of both approaches, one option discussed (but not formalized) by Xie and Oliveira, is to have the inference mode, the application mode and the checking mode in a single calculus. Then we could have a calculus that could type all the examples shown so far. Basically, this calculus would include three modes (checking, inference and application) and also three application rules: `App1`, `App2` and `AppS`. We refer to the work of Xie and Oliveira for a more detailed discussion. We give one example that showcases the power of the three different rules in combination.

$$\texttt{let}\ z = 1\ \texttt{in}\ (\lambda f.\ f\ 1 : ((I \to I) \to I))\ ((\lambda x.\ \lambda y.\ y)\ z)$$

`AppS` allows writing let-expressions as sugar to applications and without annotations (when the arguments can be inferred). The rule `App1` allows the annotation of functions to be used to check the arguments. Moreover, `App2` allows unannotated applications, such as $(\lambda x.\ \lambda y.\ y)\ z$, to be checked.

Backtracking. Unfortunately, now there would be an even worse problem with backtracking. We have 3 different rules for applications, with none of the rules being better than the others. In the worst case, we have to try the 3 rules for every application. This would be quite inefficient in practice. While using just one of the rules in an implementation addresses the efficiency problem, it gives up the expressiveness afforded by the other rules, requiring programs with more annotations or the addition of new constructs and typing rules. If we use `App1` only, then we need to write:

$$((\lambda z.\ (\lambda f.\ f\ 1 : (I \to I) \to I)\ (((\lambda x.\ \lambda y.\ y) : I \to I \to I)\ z)) : I \to I)\ 1$$

Here we desugar the `let` expression into an explicit lambda, so that we can annotate the desugared lambda. In addition, we would need to introduce an annotation in $(\lambda x.\ \lambda y.\ y)\ z$. Alternatively, we could introduce `let` expressions directly together with corresponding typing rules, to address the issue with $\texttt{let}\ z = 1\ \texttt{in}\ldots$, and also change the term $(\lambda x.\ \lambda y.\ y)\ z$ to a let expression. However, as we shall discuss in Section 2.3, introducing `let` expressions still has some issues.

2.3 Binding Constructs

As Dunfield and Krishnaswami [2022] observe, binding constructs, such as `let` expressions or eliminators for sums introduce some additional complications for bidirectional typing. The Pfenning recipe to derive bidirectional rules applies to introduction and elimination forms, but let-expressions are neither. So, the first challenge is to decide how to bidirectionalize let expressions:

$$\frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma, x : A \vdash e_2 : B}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : B}\ \texttt{Let}$$

It is unclear to further determine the direction of the second premise and conclusion. The suggested approach is to have two variants of the rule, to type check more programs.

$$\frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma, x : A \vdash e_2 \Rightarrow B}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Rightarrow B}\ \texttt{LetInf} \qquad \frac{\Gamma \vdash e_1 \Rightarrow A \qquad \Gamma, x : A \vdash e_2 \Leftarrow B}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \Leftarrow B}\ \texttt{LetChk}$$

For the example mentioned above, rewritten into a let-binding form:

$$\texttt{let}\ z = 1\ \texttt{in}\ (\lambda f.\ f\ 1 : ((I \to I) \to I))\ (\texttt{let}\ x = z\ \texttt{in}\ \lambda y.\ y)$$

the outside `let` uses `LetInf` rule to infer its result. The inside `let` uses `LetChk` to propagate the type $I \to I$ to the body. Rule `LetChk` is similar to `App2`, but specialized to `let` expressions.

Unclear annotatability. Dunfield and Pfenning [2004] suggested that annotations are only needed at redexes. However, Dunfield and Krishnaswami [2022] noted that this is not true in the presence of binding constructs, even with two rules for binding constructs. For instance, consider a type system with two `let` rules and only the `App1` rule. Now consider the (unannotated) expression:

$$(\texttt{let}\ x = 1\ \texttt{in}\ \lambda f.\ f\ x)\ (\lambda x.\ x)$$

In this case, we must either put annotations on the body of `let`, then use `LetInf` to infer the type of the `let` expression, or around the `let` expression to use the contextual information to check its body. Thus, for some bidirectional type systems with binding constructs, it may not be easy to give programmers a clear guideline as to where to place annotations.

2.4 Subtyping

In calculi with subtyping, an important idea is that the subsumption rule is used to encapsulate the uses of subtyping in a single place. This is commonly used in type assignment systems (TASs), and also in (simple) bidirectional type systems. The bidirectional rule for subtyping is analogous to rule Sub in Fig 1, with type equality replaced with subtyping.

$$\frac{\Gamma \vdash e \Rightarrow A \qquad A <: B}{\Gamma \vdash e \Leftarrow B}\ \mathsf{Sub}$$

However, in the presence of more complex subtyping relations, the bidirectional subsumption rule is not powerful enough to encapsulate many of the desired uses of subtyping. For instance, intersection types [Barendregt et al. 1983; Coppo et al. 1981; Pottinger 1980; Reynolds 1991] are used to assign multiple types to one expression. Type systems with intersection types often use three subtyping rules to characterize this idea:

$$\frac{A <: C}{A \,\&\, B <: C}\ \mathsf{SAndL} \qquad \frac{B <: C}{A \,\&\, B <: C}\ \mathsf{SAndR} \qquad \frac{A <: B \qquad A <: C}{A <: B \,\&\, C}\ \mathsf{SAnd}$$

If we use rule App1, and the bidirectional subsumption rule, then some programs that we expect to type check, would no longer type check. For instance, $(\lambda f.\, f\ 1) : ((I \to I) \,\&\, (F \to F)) \to I$ (F stands for the `Float` type) is a higher order function that expects an overloaded function with the intersection type $(I \to I) \,\&\, (F \to F)$. This function cannot type check using the App1 rule:

$$\frac{f : (I \to I) \,\&\, (F \to F) \vdash f \Rightarrow (I \to I) \,\&\, (F \to F) \qquad f : (I \to I) \,\&\, (F \to F) \vdash 1 \Leftarrow ?}{f : (I \to I) \,\&\, (F \to F) \vdash f\ 1 \Rightarrow ?}\ \mathtt{App1Fail}$$

In the rule App1, the function e_1 is expected to have function type $A \to B$, instead of the intersection type in this case. Thus, rule App1 will reject this program. In contrast, the equivalent TAS would accept the program above. A possible solution is to modify the application rule as follows:

$$\frac{\Gamma \vdash e_1 \Rightarrow A \qquad A <: B \to C \qquad \Gamma \vdash e_2 \Leftarrow B}{\Gamma \vdash e_1\ e_2 \Rightarrow C}\ \mathsf{App}$$

This rule allows for a more flexible form of application, that solves our problem, since it accounts for subtyping and for types that are subtypes of functions. However, there are two issues with this rule. The first one is that the rule gives up the principle of encapsulating uses of subtyping in the subsumption rule. The second one is that the rule is non-algorithmic (since the type B is guessed). The second issue is solvable by creating specialized relations that avoid guessing B and thus can be implemented in an algorithm. This is currently the standard solution for this problem in calculi with bidirectional typing. In calculi with intersection types, a selection/matching judgment is introduced in the application rules [Davies and Pfenning 2000; Huang et al. 2021]:

$$\frac{\Gamma \vdash e_1 \Rightarrow A \qquad A \triangleright B \to C \qquad \Gamma \vdash e_2 \Leftarrow B}{\Gamma \vdash e_1\ e_2 \Rightarrow C}\ \mathsf{App}$$

The selection judgment $A \triangleright B \to C$ simply tries to select one instance from different branches of intersection types. The selection judgment should, ideally, be sound and complete to $A <: B \to C$. Although this approach addresses the algorithmic problem, it requires modifying the standard

application rule and coming up with a new specialized rule and a new auxiliary judgment for applications. Similar solutions exist for other subtyping relations, such as relations with union types [Rioux et al. 2023], polymorphic types [Dunfield and Krishnaswami 2013; Zhao et al. 2019], etc. However, it would be preferable to use the standard application rule and delegate dealing with subtyping to a stronger subsumption rule, thus avoiding the need for specialized relations.

2.5 Problem Statement and Paper Roadmap

Bidirectional typing provides great value for money: it is a simple, lightweight approach that can scale up to many different type systems and features. In practice, it eliminates the need for many obvious annotations, while requiring only a reasonable amount of annotations. However, as seen in this section, it does have some limitations. In particular, we identified three concrete problems:

(1) **Trade-off between expressive power and backtracking.** As we have seen in Sections 2.1 and 2.2, there is no single application rule that is always better. Choosing one of the rules only limits the expressive power (i.e. it results in more required annotations). Choosing all the rules raises efficiency concerns since backtracking seems to be required.
(2) **Unclear annotatability and rule duplication.** In the presence of certain features, and in particular binding constructs, there is no simple guideline for how to identify where annotations are needed. Moreover, often multiple versions of a rule are needed.
(3) **Inexpressive subsumption.** The bidirectional subsumption rule lacks expressive power. Many type systems with subtyping require changes in other rules, to accommodate for the lack of expressive power of bidirectional subsumption. In addition, new auxiliary relations, which provide specialized forms of subtyping, are often needed.

We would like to retain the key advantages of bidirectional typing, while improving on the three points above. As we shall see, contextual typing provides a generalization of bidirectional typing that largely realizes this goal. Sections 3 and 4 will show how to address (1) and (2) first in a simplified setting. Section 5 will show how to deal with (3) and present a more complete type system with subtyping.

3 Quantitative Type Assignment Systems

In this section, we introduce Quantitative Type Assignment Systems (QTASs). QTASs are variants of type assignment systems enriched with a notion of counters, which generalize modes in bidirectional typing. QTASs serve as an intermediate step in between traditional type assignment systems and algorithmic formulations. A QTAS guides the formulation of the algorithmic system and aids in its metatheory. Unlike type assignment systems, QTASs precisely specify where type annotations are needed. Moreover, they *quantify* the amount of type information that a term needs in order to type check. As we shall see, with QTASs, we can have a type system that includes all the expressive power of the three application rules discussed in Section 2. In Section 4 we will present a *non-backtracking* algorithmic formulation that can be used to implement our most expressive QTAS in this section.

3.1 Quantitative Type Assignment Systems for STLC

We present several variants of QTAS for STLC in this section to illustrate the key ideas and to establish a connection to standard bidirectional typing and the let arguments go first approaches.

Syntax. The syntax of types and typing environments is the same for all the variants, but expressions and counters change according to each QTAS. For expressions the key difference is whether annotated expressions ($e : A$) are supported or not. We define types and environments below. Metavariables A, B, C, D range over types. Types are integer types and function types. A typing environment is a sequence of variables associated with their types.

$\boxed{\Gamma \vdash_n e : A}$ (Under Γ, expression e needs n contextual information to have type A.)

$$\textsf{DLit}\ \frac{}{\Gamma \vdash_0 i : \texttt{Int}} \qquad \textsf{DVar}\ \frac{x : A \in \Gamma}{\Gamma \vdash_0 x : A} \qquad \textsf{DAnn}\ \frac{\Gamma \vdash_\infty e : A}{\Gamma \vdash_0 (e : A) : A} \qquad \textsf{DLam}\ \frac{\Gamma, x : A \vdash_\infty e : B}{\Gamma \vdash_\infty \lambda x.\, e : A \to B}$$

$$\textsf{DApp1}\ \frac{\Gamma \vdash_0 e_1 : A \to B \qquad \Gamma \vdash_\infty e_2 : A}{\Gamma \vdash_0 e_1\ e_2 : B} \qquad \textsf{DApp2}\ \frac{\Gamma \vdash_\infty e_1 : A \to B \qquad \Gamma \vdash_0 e_2 : A}{\Gamma \vdash_\infty e_1\ e_2 : B} \qquad \textsf{DSub}\ \frac{\Gamma \vdash_0 e : A \qquad A = B}{\Gamma \vdash_\infty e : B}$$

Fig. 3. STLC with all-or-nothing counters.

Types	$A, B, C, D ::= \texttt{Int} \mid A \to B$
Typing Environments	$\Gamma ::= \cdot \mid \Gamma, x : A$

3.2 All-or-Nothing Counters

The first QTAS that we are going to show is the one with all-or-nothing counters. We define its expressions and counters below. This QTAS is equivalent to the standard bidirectional typing rules presented in Fig. 1 extended with the App2 rule, and with an inference rule for integers.

Expressions	$e ::= i \mid x \mid \lambda x.\, e \mid e_1\ e_2 \mid e : A$
Counters	$n ::= 0 \mid \infty$

Expressions consist of integers, variables, lambdas, applications and annotated terms. In this version of a QTAS, counters are degenerate and can only be nothing (0) or everything (∞). We use counters to represent how much type information is needed, from the surrounding context, when type checking expressions. The two counters represent two extremes: 0 means that no type information is needed to type a term; ∞ means that all type information is needed to type a term.

Typing and comparison with modes. We present typing in Fig. 3. The form of the typing judgement is $\Gamma \vdash_n e : A$, which means that under environment Γ, expression e is typeable with A assuming n contextual information about A. Readers familiar with bidirectional typing may interpret two instances of counters as modes: counter 0 is the inference mode and counter ∞ is the checking mode. In other words, for this first formulation of a QTAS, the rules that we present in Fig. 3 are almost the same as those in Fig. 1. The main difference is notational: we simply use counters instead of modes, which is of course a rather superficial difference for this version.

There are two other differences to the rules in Fig. 1. Firstly, we include a version of the rule App2 (rule DApp2) as well, because we wish to capture all mode-correct rules for applications in standard bidirectional typing. Secondly, the DLit rule, when interpreted bidirectionally, is in inference mode, rather than in checking mode. Using 0 as the counter for the DLit is consistent with the interpretation of the counter, which measures how much information is needed from the surrounding context. For integers no information is needed. Finally, although the DSub rule is the same as in bidirectional typing, it is worthwhile reinterpreting the rule from the point of view of a QTAS. This rule states that a term that needs no information to be typed can be interpreted as a term that requires more information to be typed. In other words, additional contextual information does not affect typeability. The soundness and completeness theorems to the type system in Fig. 1, extended with an inference rule for integers and with the App2 rule are straightforward.

Theorem 3.1 (Soundness).

- *If* $\Gamma \vdash_0 e : A$ *then* $\Gamma \vdash e \Rightarrow A$.
- *If* $\Gamma \vdash_\infty e : A$ *then* $\Gamma \vdash e \Leftarrow A$.

Theorem 3.2 (Completeness).

- *If* $\Gamma \vdash e \Rightarrow A$ *then* $\Gamma \vdash_0 e : A$.
- *If* $\Gamma \vdash e \Leftarrow A$ *then* $\Gamma \vdash_\infty e : A$.

$$\frac{}{\Gamma \vdash_0 i : \mathtt{Int}}\ \mathtt{DLit} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash_0 x : A}\ \mathtt{DVar} \qquad \frac{\Gamma, x : A \vdash_n e : B}{\Gamma \vdash_{(S\ n)} \lambda x.\ e : A \to B}\ \mathtt{DLam}$$

$$\frac{\Gamma \vdash_{(S\ n)} e_1 : A \to B \qquad \Gamma \vdash_0 e_2 : A}{\Gamma \vdash_n e_1\ e_2 : B}\ \mathtt{DApp}$$

Fig. 4. STLC with application counters.

3.3 Application Counters

We now present another QTAS with *application counters* that is equivalent to the type system in Fig. 2. Unlike all-or-nothing counters, application counters measure *how much* contextual type information we need to type a term. We define the syntax below and present the typing in Fig. 4.

Expressions	$e ::= i \mid x \mid \lambda x.\ e \mid e_1\ e_2$
Counters	$n ::= 0 \mid S\ n$

We have two instances of counters: nothing (0) and $S\ n$. We do not deal with annotated terms, thus $e : A$ is not included. The two interesting rules are DLam rule and DApp. In DLam, counter $S\ n$ indicates that in the function type $A \to B$, the input type A must be known from the surrounding context. Then we can use it as the type of bound variable x. In DApp, we infer the type of e_2, and increment the counter n of e_1, to express that A is known. Thus, we can infer the type $A \to B$ with the help of propagated information. The output type B will be the result type of the application.

Intuitively, the counter denotes the size of the application stack. Thus we can show a correspondence to the type system in Fig. 2 with the following theorems. What we need is to define an auxiliary judgment that connects the counter to the application stack $n \sim \Psi \sim A$. The 0 counter connects to an empty stack and any type, and $S\ n$ connects to Ψ, A and $A \to B$ inductively.

$$\frac{}{0 \sim . \sim A}\ \sim\mathtt{Empty} \qquad \frac{n \sim \Psi \sim B}{S\ n \sim \Psi, A \sim A \to B}\ \sim\mathtt{Cons}$$

If the term e can be typed with the type A with counter n and n is consistent with Ψ, it can infer A with the application stack Ψ, and vice-versa.

Theorem 3.3 (Soundness). *If* $\Gamma \vdash_n e : A$ *and* $n \sim \Psi \sim A$, *then* $\Gamma \shortmid \Psi \vdash e \Rightarrow A$.

Theorem 3.4 (Completeness). *If* $\Gamma \shortmid \Psi \vdash e \Rightarrow A$ *and* $n \sim \Psi \sim A$, *then* $\Gamma \vdash_n e : A$.

3.4 All in One

Finally, we show a QTAS that combines all-or-nothing counters and application counters. Compared with the previous QTASs, we include both annotated terms and application counters.

Expressions	$e ::= i \mid x \mid \lambda x.\ e \mid e_1\ e_2 \mid e : A$
Counters	$n ::= 0 \mid \infty \mid S\ n$
Syntactic Sugar	$\mathtt{let}\ x\ =\ e_1\ \mathtt{in}\ e_2 \triangleq (\lambda x.\ e_2)\ e_1$

We present our typing rules in Fig. 5. DLit and DVar are unsurprising. In rule DAnn the annotated term $e : A$ is typeable with type A without any additional contextual type information, if e is typeable with A with full contextual information. DLam type checks a lambda only when the counter can be decreased, which means that the information for the input type A must be available contextually. The $n-$ meta-operation is defined with two simple cases: $\infty - = \infty$; and $(S\ n) - = n$. The lambda body (e) may itself require some amount of contextual information n. For instance, the lambda term $\lambda x.\ \lambda y.\ x + y$ is typeable if the counter is at least 2: the lambda body $\lambda y.\ x + y$ requires type information for y, whereas the outer lambda requires type information for x. Rules DApp1 and DApp2 describe two situations: whether the function can infer or the argument can infer. We use the word *infer* here to represent the situation where the counter is 0. In DApp1, if the function

$$\frac{\Gamma \vdash_\infty e : A}{\Gamma \vdash_0 (e : A) : A}\ \textsf{DAnn} \qquad \frac{\Gamma, x : A \vdash_{n-} e : B}{\Gamma \vdash_n \lambda x.\, e : A \to B}\ \textsf{DLam} \qquad \frac{\Gamma \vdash_0 e_1 : A \to B \qquad \Gamma \vdash_\infty e_2 : A}{\Gamma \vdash_0 e_1\ e_2 : B}\ \textsf{DApp1}$$

$$\frac{}{\Gamma \vdash_0 i : \texttt{Int}}\ \textsf{DLit} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash_0 x : A}\ \textsf{DVar} \qquad \frac{\Gamma \vdash_0 e : A \qquad n \neq 0}{\Gamma \vdash_n e : A}\ \textsf{DSub} \qquad \frac{\Gamma \vdash_{(S\ n)} e_1 : A \to B \qquad \Gamma \vdash_0 e_2 : A}{\Gamma \vdash_n e_1\ e_2 : B}\ \textsf{DApp2}$$

Fig. 5. All-in-one QTAS for STLC.

$$\dfrac{\dfrac{\Gamma, z : I \vdash_0 z : I \qquad \dfrac{\cdots}{\Gamma, z : I, x : I \vdash_\infty \lambda y.\, y : I \to I}\ \textsf{DLam}}{\cdots \qquad \Gamma, z : I \vdash_\infty \texttt{let}\ x = z\ \texttt{in}\ (\lambda y.\, y) : I \to I}\ \textsf{Let} \qquad \Gamma, z : I \vdash_0 (\lambda f.\, f\ 1 : ((I \to I) \to I))\ (\texttt{let}\ x = z\ \texttt{in}\ (\lambda y.\, y)) : I}{\Gamma \vdash_0 1 : I \qquad \Gamma, z : I \vdash_0 (\lambda f.\, f\ 1 : ((I \to I) \to I))\ (\texttt{let}\ x = z\ \texttt{in}\ (\lambda y.\, y)) : I}\ \textsf{DApp1}}{\Gamma \vdash_0 \texttt{let}\ z = 1\ \texttt{in}\ (\lambda f.\, f\ 1 : ((I \to I) \to I))\ (\texttt{let}\ x = z\ \texttt{in}\ (\lambda y.\, y)) : I}\ \textsf{Let}$$

Fig. 6. Typing derivation for the motivating example in Section 2.2.

e_1 infers the type $A \to B$, then we have all the information available for the argument, and can use that to check whether e_2 has type A. In rule DApp2, if the argument e_2 infers the type A, we increment the counter for the function, since we have all the type information for the argument. Thus e_1 can have the type $A \to B$ with additional contextual information. Note that DApp2 merges the expressiveness of rules App2 and AppS. Thus, we only need two rules, instead of three. As before, DSub expresses that additional contextual information does not affect typeability.

We show that our all-in-one QTAS is complete with respect to both traditional bidirectional typing (including rule App2 and integer inference) and application modes. Note that we allow successor wrapping around ∞. Thus we use a relation $\preceq$ that relates modes to counters.

$$\frac{}{\Rightarrow \preceq 0}\ \preceq 0 \qquad \frac{}{\Leftarrow \preceq \infty}\ \preceq \infty \qquad \frac{\Leftarrow \preceq n}{\Leftarrow \preceq S\ n}\ \preceq \text{S}$$

Theorem 3.5 (Completeness).

- *If* $\Gamma \vdash e \Leftrightarrow A$ *and* $\Leftrightarrow \preceq n$, *then* $\Gamma \vdash_n e : A$.
- *If* $\Gamma \shortmid \Psi \vdash e \Rightarrow A$ *and* $n \sim \Psi \sim A$, *then* $\Gamma \vdash_n e : A$.

Encoding let expressions. Like the approach by Xie and Oliveira [2018], we can encode `let` expressions as syntactic sugar. The following derivation shows the derivable typing rule for let expressions (from the syntactic sugar):

$$\frac{\Gamma \vdash_0 e_1 : A \qquad \Gamma, x : A \vdash_n e_2 : B}{\Gamma \vdash_n \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : B}\ \textsf{Let} \qquad \frac{\dfrac{\Gamma, x : A \vdash_n e_2 : B}{\Gamma \vdash_{S\ n} \lambda x.\, e_2 : A \to B}\ \textsf{DLam} \qquad \Gamma \vdash_0 e_1 : A}{\Gamma \vdash_n (\lambda x.\, e_2)\ e_1 : B}\ \textsf{DApp2}$$

On the left we present the derived rule and on the right we desugar the let expression to show how the rule can be derived. This rule subsumes the two bidirectional typing rules for let expressions presented in Section 2.3. The inference rule corresponds to n being 0 and the checking rule corresponds to n being ∞. Moreover, n can also be $S\ n'$, which has no correspondence in the bidirectional rules. Figure 6 illustrates the expressiveness of the QTAS with our example in Section 2.2, which requires the power of the three application rules.

A remark and a closer look at counters. An important remark here is that the STLC allows for a simple formulation of counters. As we shall see in Section 5, in more complex calculi, we may need

a more fine-grained definition of counters. It is useful to have a closer look at the role of counters to better understand how counters are used by a QTAS.

Counters in Fig. 5 track the number of times that the DApp2 rule needs to be applied to obtain enough type information from the surrounding context. In turn this implies that counters track the number of arguments that must have inferred types (since DApp2 requires that we must be able to infer types for arguments). For instance consider the term $(\lambda x.\ f)\ 1\ (\lambda x.\ x)\ 2$, where f has type $(I \to I) \to I \to I$ in the typing environment. The typing derivation for this term is:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\Gamma \vdash_{(S\ 0)} \lambda x.\ f : I \to (I \to I) \to I \to I}\ \texttt{Lam} \quad \Gamma \vdash_0 1 : I}{\Gamma \vdash_0 (\lambda x.\ f)\ 1 : (I \to I) \to I \to I}\ \texttt{DApp2} \quad \Gamma \vdash_\infty \lambda x.\ x : I \to I}{\Gamma \vdash_0 (\lambda x.\ f)\ 1\ (\lambda x.\ x) : I \to I}\ \texttt{DApp1} \quad \Gamma \vdash_\infty 2 : I}{\Gamma \vdash_0 (\lambda x.\ f)\ 1\ (\lambda x.\ x)\ 2 : I}\ \texttt{DApp1}$$

For typing this term it is necessary that the type of the argument 1 is inferable (which is the case). In other words we must use DApp2 to type $(\lambda x.\ f)\ 1$. When typing the abstraction $\lambda x.\ f$ the QTAS uses a counter $S\ 0$, to signal that the lambda expression can only be typed in a context with an inferable argument. The other two arguments ($\lambda x.\ x$ and 2) will be checked against $I \to I$ and I, respectively, by using DApp1. Importantly, the arguments checked with DApp1 do not really influence the types used in the typing derivation. While this is true for STLC, it is not true in general. In some type systems, especially those with subtyping, the checkable arguments that are typed with DApp1 may influence the typing derivation in significant ways. In that case, it is also important to track the number of uses of DApp1 or checkable arguments (in addition to the number of inferable arguments). We will come back to this point in Section 5.

3.5 Comparison and Correspondence to Type Assignment Systems

Differently from type assignment systems, QTASs use counters to quantify needed contextual information, and restrict typing derivations. Still, QTASs are (in general) non-algorithmic. For instance, in rule DLam (from Fig. 5), a counter $S\ n$ in the conclusion tells us that we have type information for at least one argument, but does not specify what the type is. We could try to interpret the typing judgment algorithmically as a function that takes the typing environment Γ, the expression e and the counter n as inputs, and returns the type A as an output. However, it should be clear that the counter alone does not provide enough information to determine the type information available from the context and compute the output type. The role of the counter in a QTAS is merely to constrain which rules can be used, but types are still being guessed as in a TAS. For example, if we have an identity function with a non-zero counter, this function can be typed with multiple types.

$$\dfrac{\Gamma, x : \texttt{Int} \vdash_0 x : \texttt{Int}}{\Gamma \vdash_{(S\ 0)} \lambda x.\ x : \texttt{Int} \to \texttt{Int}}\ \texttt{DLam} \qquad \dfrac{\Gamma, x : (\texttt{Int} \to \texttt{Int}) \vdash_0 x : \texttt{Int} \to \texttt{Int}}{\Gamma \vdash_{(S\ 0)} \lambda x.\ x : (\texttt{Int} \to \texttt{Int}) \to (\texttt{Int} \to \texttt{Int})}\ \texttt{DLam}$$

Completeness and annotatability. TAS often work on unannotated expressions, guessing all types. For calculi with undecidable type systems, it is impossible to find algorithms that directly infer types for all unannotated expressions. Thus, algorithms often need to deal with annotated expressions instead. Unlike a TAS, a QTAS requires *annotations* in a program. Only programs with sufficient annotations will type-check. Therefore a QTAS can serve as a specification of a type system informing where annotations are needed to type check programs.

The *completeness* of a QTAS with regard to a TAS tells us that a term in a TAS can be typed in a QTAS after adding some annotations. Dunfield and Krishnaswami [2022] also call this result

$$\frac{}{\Gamma \vdash i : \mathtt{Int} \rightsquigarrow i}\ \mathtt{ELit} \qquad \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A \rightsquigarrow x}\ \mathtt{EVar} \qquad \frac{\Gamma, x : A \vdash e : B \rightsquigarrow e'}{\Gamma \vdash \lambda x.\, e : A \to B \rightsquigarrow \lambda x.\, e'}\ \mathtt{ELam}$$

$$\frac{\mathsf{need}\ e_1 = 0 \vee \mathsf{need}\ e_2 = 0 \qquad \Gamma \vdash e_1 : A \to B \rightsquigarrow e_1' \qquad \Gamma \vdash e_2 : A \rightsquigarrow e_2'}{\Gamma \vdash e_1\ e_2 : B \rightsquigarrow e_1'\ e_2'}\ \mathtt{EApp1}$$

$$\frac{\mathsf{need}\ e_1 = S\ n_1 \wedge \mathsf{need}\ e_2 = S\ n_2 \qquad \Gamma \vdash e_1 : A \to B \rightsquigarrow e_1' \qquad \Gamma \vdash e_2 : A \rightsquigarrow e_2'}{\Gamma \vdash e_1\ e_2 : B \rightsquigarrow e_1'\ (e_2' : A)}\ \mathtt{EApp2}$$

Fig. 7. Elaboration from a TAS to a QTAS, with one possible annotation strategy.

annotatability, which we prefer as it is more descriptive. For instance, we can prove the following annotatability lemma between a TAS formulation of STLC and the QTAS in Fig. 5:

Theorem 3.6 ((Weak) Annotatability of QTAS). *If* $\Gamma \vdash e : A$, *then* $\exists\, e', \Gamma \vdash_0 e' : A$ *and* e *is the type erasure of* e'.

The standard annotatability theorem, which we will refer to as *weak annotatability*, shows that our QTAS is complete to a TAS by inserting annotations on the unannotated terms. However, it does not tell us *where* type annotations are needed. In bidirectional typing, by strictly following the Pfenning recipe, we can obtain an informal guideline of only putting annotations on redexes. However this guideline, after adding more constructs and more expressive rules, becomes less clear [Dunfield and Krishnaswami 2022] (see also Section 2.3).

Strong annotatability of a QTAS. With a QTAS we can prove a stronger annotatability theorem that also tells us where annotations are needed and *formalizes* guidelines for how to annotate a program. To formalize strong annotatability, we first introduce a function need, which specifies the type information needed by an (unannotated) term to type check in a QTAS.

$$\mathsf{need}\ (\lambda x.\, e) = S\ (\mathsf{need}\ e) \qquad \mathsf{need}\ i = 0$$

$$\mathsf{need}\ (e_1\ e_2) = \begin{cases} 0, & \text{if need } e_1 = 0 \\ n, & \text{if need } e_1 = S\ n \end{cases} \qquad \mathsf{need}\ x = 0$$

For a lambda, we need type information for the argument, and the body itself may need some type information, so we need one more than the body. For an application, if the function e_1 needs no type information (i.e. e_1 is an inferable term), then the whole application does not need type information to type check. If the function e_1 needs some amount of type information $S\ n$, the application will need n, since the only rule that can be used in this case is DApp2, so the argument e_2 must be inferable, thus providing information for the argument of the function.

With need we define a (type-directed) elaboration from a TAS to a QTAS in Fig. 7. In this elaboration annotations are inserted where needed. The first three rules are straightforward. Only the two application rules are interesting. EApp1 describes the case where applications do not need annotations: if e_1 needs no type information, then we can apply DApp1 to type check this expression; if e_2 needs no type information, then we can apply DApp2. EApp2 describes the case where an application needs additional type information: if both e_1 and e_2 cannot be inferred (for example $(\lambda x.\, x)\ (\lambda x.\, x)$), we then choose to annotate the argument e_2 with the type A. With this elaboration we can prove the stronger annotatability theorem:

Theorem 3.7 (Strong Annotatability).

(1) If $\Gamma \vdash e : A \rightsquigarrow e'$, *then* $\Gamma \vdash_{(\mathsf{need}\ e)} e' : A$. *(2) If* $\Gamma \vdash e : A \rightsquigarrow e'$, *then* $\Gamma \vdash_0 (e' : A) : A$.

The first result (1) states that if the term e is well-typed in the TAS and elaborated to e' with annotations inserted, then e' is well-typed in the QTAS, provided that enough contextual type

information is given. The second result (2) states that, with a top-level annotation we obtain an inferable term. This theorem formalizes a simple annotatability guideline for programmers. Basically, programmers need to:

- provide a top-level type annotation for a term if the term needs type information (for instance the full term is a lambda) (2);
- and, for applications where neither the function or the argument can be inferred, provide an annotation for the argument (1).

Since `let` expressions $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ are encodable via applications, from the guideline above we can deduce that if e_1 is not inferable then we will need to add an annotation for e_1.

Note that the annotation guideline is not unique: we could have different ways to annotate programs. For instance, instead of annotating the argument, we can annotate the function e_1. This leads to the following elaboration rule:

$$\frac{\mathsf{need}\ e_1 \equiv S\ n_1 \wedge \mathsf{need}\ e_2 \equiv S\ n_2 \qquad \Gamma \vdash e_1 : A \to B \rightsquigarrow e_1' \qquad \Gamma \vdash e_2 : A \rightsquigarrow e_2'}{\Gamma \vdash e_1\ e_2 : B \rightsquigarrow (e_1' : A \to B)\ e_2'}\ \texttt{EApp3}$$

Rules `EApp2` and `EApp3` provide two different alternative annotatability strategies/guidelines. We can replace `EApp2` by `EApp3`, or include both rules and the strong annotatability lemma still holds.

To conclude we show how to use our guideline to annotate the example in Section 2.3. In essence, since in the application neither the function nor the argument are inferable, we add an annotation to the argument, as prescribed by our guideline:

$$(\texttt{let}\ x = 1\ \texttt{in}\ \lambda f.\ f\ x)\ (\lambda x.\ x : I \to I)$$

4 Syntax-Directed Algorithmic Type System

In this section, we provide a *syntax-directed* algorithmic type system that avoids backtracking and implements the QTAS in Section 3.4. We start with the key idea behind the algorithm, and then show the full formalization. Finally, we will discuss its metatheory: decidability, soundness and completeness with respect to the QTAS.

4.1 Towards a Non-Backtracking Algorithm

A possible way to naively implement the QTAS discussed in Section 3.4, would be to use the approach discussed in Section 2.2. Then we would have the three application rules (`App1`, `App2` and `AppS`) and a type system combining three modes (inference, checking and application). Using backtracking, we could try all the 3 rules. A key drawback of this approach is performance, since backtracking could be very costly in practice.

Our goal is to have the same expressive power as the QTAS without using backtracking. When designing the QTAS we already saw that `App2` and `AppS` can be combined into a single rule (`DApp2`), which removes some of the overlap between the application rules. However, we still have two application rules and there are some types being guessed. Thus, in order to find a non-backtracking algorithmic formulation, we must overcome these two problems first. To do so, it is helpful to analyse the root cause of backtracking. For this we first identify when we need to use `DApp1` and `DApp2`. Let us consider two special cases in an application $e_1\ e_2$: the function e_1 is a lambda abstraction $\lambda x.\ e$; or the function e_1 is a variable x. For these two cases we can clearly identify which rule is better:

$$\frac{\Gamma \vdash_{(S\ n)} \lambda x.\ e : A \to B \qquad \Gamma \vdash_0 e_2 : A}{\Gamma \vdash_n (\lambda x.\ e)\ e_2 : B}\ \texttt{DApp2} \qquad \frac{\Gamma \vdash_0 x : A \to B \qquad \Gamma \vdash_\infty e_2 : A}{\Gamma \vdash_0 x\ e_2 : B}\ \texttt{DApp1}$$

When e_1 is a lambda abstraction $\lambda x.\ e$, `DApp1` will never succeed because $\lambda x.\ e$ is not typeable without contextual information. Thus, the only rule that we can use in this case is `DApp2` which, if

it successfully infers a type for e_2, can then try to infer the type for the abstraction. When e_1 is a variable x it is not too difficult to see that DApp1 is always a better rule to use, since the type of variables can always be inferred. So using DApp1 instead of DApp2 will lead to strictly more successful typing derivations. One other case that is similar to variables is when e_1 is an annotation expression $e : A$. In this case, DApp1 is also the best choice. For the cases that we have analysed it seems that we can pick the best application rule, based on the syntactic structure of e_1. This is true, but e_1 can, in the general case, have more complex shapes, and those have to be dealt with as well.

Application consumer. An *application consumer* is either a variable, a lambda or an annotated term. The three cases in an application consumer capture the three cases that we analysed so far. For an application $e_1\ e_2$ what we are looking for is the application consumer for e_2. That is, the term that will eventually consume the contextual information for the argument. In direct applications, such as $(\lambda x.\ x)\ e_2$, the application consumer is just e_1 ($\lambda x.\ x$ in this case). In the general case the consumer may lie deep within e_1. Consider the application $((\lambda x.\ y)\ 1)\ (\lambda z.\ z)$. Here y is a variable, with the type $(I \to I) \to I$ in the typing environment. Note that this term is typeable if we use DApp1 for the outer application and DApp2 for the inner application. In this case y is the consumer that determines which application rule to use to type check the outer application (for the argument $\lambda z.\ z$). We should choose DApp1, since the application consumer is a variable (y). Conversely, for the inner application we should use DApp2 since the application consumer is $\lambda x.\ y$. A second example is $(\lambda x.\ \lambda y.\ x + y)\ 1\ 2$, where the consumer for 2 is $\lambda y.\ x + y$ and the consumer for 1 is $(\lambda x.\ \lambda y.\ x + y)$.

To design an efficient algorithm the notion of application consumer is useful, since the consumer determines the best application rule to use. Therefore, once we know what is the best rule, we do not need to try any other rules. A backtracking algorithm does not attempt to choose the best rule to apply. Instead, it simply blindly tries each rule and, if some rule fails, it tries another one. To find the best rule to apply, one possibility is to analyse the structure of the application to decide which rule to apply. In other words we could look into an application $e_1\ e_2$ and find the consumer for e_2. Then, using the information about the consumer, we could decide which rule use in the application. Nonetheless this approach still requires us to do multiple traversals on e_1: traversing e_1 to determine the syntactic form of the application consumer; and traversing e_1 again to actually type check the application. However, it is possible to do better, and traverse e_1 only one time.

4.2 Key Idea: Teleporting Typing Judgements

Knowing about the syntactic form of the application consumer determines what application rule is best to use, but it does not determine whether typing will be successful or not. For instance, we could have the ill-typed application $(\lambda x.\ x + 1)$ `true`, for which we could determine that the best rule to use is DApp2, but typing would nonetheless fail. The key idea in our algorithm is to bring the arguments and the corresponding application consumer together, so that once we find the consumer, we immediately check whether the application succeeds or not. To do this, in an application $e_1\ e_2$, we delay the typing of e_2 until we encounter the application consumer in the typing derivation. To visualize this idea consider the following (pseudo) derivation:

$$\dfrac{\dfrac{\dfrac{\dots \quad \boxed{\Gamma, x : I, y : I \vdash 2 : I}}{\Gamma, x : I \vdash \lambda y.\ x + y : I \to I}\ \textsf{Lam} \qquad \boxed{\Gamma, x : I \vdash 1 : I}}{\Gamma \vdash \lambda x.\ \lambda y.\ x + y : I \to I \to I}\ \textsf{Lam} \qquad \boxed{\Gamma \vdash 1 : I}}{\Gamma \vdash (\lambda x.\ \lambda y.\ x + y)\ 1 : I \to I}\ \textsf{App} \qquad \boxed{\Gamma \vdash 2 : I}}{\Gamma \vdash ((\lambda x.\ \lambda y.\ x + y)\ 1)\ 2 : I}\ \textsf{App}$$

Normally, when encountering an application $e_1\ e_2$, we would attempt to check the typing of e_2

Types	$A, B, C, D ::= \texttt{Int} \mid A \to B$
Expressions	$e ::= i \mid x \mid \lambda x.\, e \mid e_1\ e_2 \mid e : A$
Typing Environments	$\Gamma ::= \cdot \mid \Gamma, x : A$
Surrounding Contexts	$\Sigma ::= \square \mid A \mid \boxed{e} \mapsto \Sigma$
Generic Consumers	$g ::= i \mid x \mid e : A$

Fig. 8. Syntax for the algorithmic type system.

directly in a premise of the application rule for $e_1\ e_2$. Instead, what we want is to transport the typing premise for e_2 into the rule that deals with the application consumer. We call this process of moving the typing premise for e_2 into the application consumer *teleportation*. For instance, in the derivation above, we teleport $\Gamma \vdash 2 : I$ into a premise of the corresponding consumer $\lambda y.\, x + y$. Similarly, we teleport the typing premise $\Gamma \vdash 1 : I$ into a premise of $\lambda x.\, \lambda y.\, x + y$.

Teleportation also works for the other 2 types of application consumers: variables and annotated expressions. In the algorithm shown next, this idea can be implemented by having an auxiliary form of context, which captures the surrounding context of an expression.

4.3 Algorithmic Type System

Syntax. The algorithmic type system shares the same syntax of types, expressions and typing environments as the QTAS in Section 3.4. The differences are two more syntactic categories: surrounding contexts and generic consumers. The full syntax is shown in Fig. 8. A surrounding context (or context for short) captures the information that is in context for the current expression. A surrounding context can be empty ($\square$), which means that the context provides no information. A context can be a full type A, which comes from type annotations, or known type information. More interestingly, a context can also be a sequence of expressions e, which denote arguments to applications, followed by more contextual information. Intuitively, expressions in the context represent deferred type checking tasks of applications, to be carried out when the application consumer is found in the derivation.

As we have seen in Section 4.1 an important concept is the notion of application consumers. Application consumers consume contextual information, and are especially interesting because, in particular, they consume the contextual information about application arguments. However, *any* expression can be a consumer of contextual information, since contextual information can also be type information coming from type annotations. For instance, in the annotated expression $1 : \texttt{Int}$, the surrounding context of 1 is `Int`, and 1 would consume that contextual type information. Notice though, that it does not make sense for 1 to consume argument information, since 1 cannot be applied (thus 1 is not an application consumer). Some kinds of expressions/consumers need to handle contextual information in a special way, but some other expressions handle contextual information in a generic way, via the subsumption rule. We call such expressions *generic consumers*. Generic consumers in our calculus are integers, annotated expressions and variables.

Surrounding context and elimination forms. An important question that a language designer may ask at this point is: how do we determine the information that needs to be tracked in the surrounding context in the general case? While in the previous section we have already motivated the need for tracking arguments of applications, what if we extend the language with new constructs? A general answer to this question is that we need to look at *elimination* forms. In the STLC there is only one elimination form: applications $e_1\ e_2$. The arguments of applications provide information that is helpful for the application consumers, which include the corresponding introduction form (lambdas), as well as variables and annotations. The information from the arguments can then

$\boxed{\Gamma \vdash \Sigma \Rightarrow e \Rightarrow A}$ (Typing: under environment Γ and context Σ, expression e infers type A.)

$$\frac{}{\Gamma \vdash \square \Rightarrow i \Rightarrow \texttt{Int}}\ \texttt{ALit} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash \square \Rightarrow x \Rightarrow A}\ \texttt{AVar} \qquad \frac{\Gamma \vdash A \Rightarrow e \Rightarrow B}{\Gamma \vdash \square \Rightarrow e : A \Rightarrow A}\ \texttt{AAnn}$$

$$\frac{\Gamma \vdash \boxed{e_2} \mapsto \Sigma \Rightarrow e_1 \Rightarrow A \to B}{\Gamma \vdash \Sigma \Rightarrow e_1\ e_2 \Rightarrow B}\ \texttt{AApp} \qquad \frac{\Gamma \vdash \square \Rightarrow e_2 \Rightarrow A \qquad \Gamma, x : A \vdash \Sigma \Rightarrow e \Rightarrow B}{\Gamma \vdash \boxed{e_2} \mapsto \Sigma \Rightarrow \lambda x.\ e \Rightarrow A \to B}\ \texttt{ALam2}$$

$$\frac{\Gamma, x : A \vdash B \Rightarrow e \Rightarrow C}{\Gamma \vdash A \to B \Rightarrow \lambda x.\ e \Rightarrow A \to C}\ \texttt{ALam1} \qquad \frac{\Gamma \vdash \square \Rightarrow g \Rightarrow A \qquad \Sigma \neq \square \qquad A \approx \Sigma}{\Gamma \vdash \Sigma \Rightarrow g \Rightarrow A}\ \texttt{ASub}$$

$\boxed{\Gamma \vdash A \approx \Sigma}$ (Matching: under environment Γ, type A is matched by context Σ.)

$$\frac{}{\Gamma \vdash A \approx \square}\ \texttt{SubEmpty} \qquad \frac{}{\Gamma \vdash A \approx A}\ \texttt{SubType} \qquad \frac{\Gamma \vdash A \Rightarrow e \Rightarrow C \qquad \Gamma \vdash B \approx \Sigma}{\Gamma \vdash A \to B \approx \boxed{e} \mapsto \Sigma}\ \texttt{SubTerm}$$

Fig. 9. Algorithmic typing and matching for the STLC.

provide the consumers with enough type information for checking typeability. Thus we need to analyse the elimination forms in the language and identify the information that is needed for aiding the consumers to establish typeability. Section 5 shows how this idea extends to record projections.

Typing. We show the full rules for algorithmic typing in Fig. 9. Typing has the form $\Gamma \vdash \Sigma \Rightarrow e \Rightarrow A$, which is interpreted as: under typing environment Γ and a surrounding context Σ, the expression e infers the type A. Under this interpretation Γ, Σ and e are inputs, and the type A is an output, determined by the three input parameters of the typing relation.

We can group rules `ALit`, `AVar` and `AAnn` together: they all infer the type without needing any contextual information. Note that these rules cover all the generic consumer expressions. The empty surrounding context $\square$ expresses that no contextual information is needed. `ALit` and `AVar` are unsurprising. `AAnn` infers the type A from its annotation and the A will become surrounding context information to infer the expression e.

There is a single rule for applications, unlike in the QTAS. Rule `AApp` simply adds the argument e_2 to the surrounding context Σ. Using this extended context we then infer e_1's type, and obtain the function type $A \to B$. The type B will be the result type of the application $e_1\ e_2$. Conversely, we now have two rules for lambda expressions, unlike the QTAS, which has a single rule. Rules `ALam1` and `ALam2` cover two cases when inferring the type of a lambda expression. The first case (rule `ALam1`) is that the context is a type $A \to B$, which means that the lambda is fully annotated. We use the type A as the type of bound variable x and add B to the context to help infer the lambda body. After we obtain the type C, we then infer the type $A \to C$. The second case (rule `ALam2`) is when the first entry in the context is an argument expression e_2. In this case, we infer the type of e_2 and obtain the type A. The type A is used as the type for the lambda variable x, and we further infer the type lambda body with the context Σ. Once we get the type B, the final inference result for the lambda expression is $A \to B$. Importantly, note that the two rules do *not overlap* since the syntactic form for the context is different. So the rules are syntax-directed.

Subsumption and the matching judgment. The subsumption rule `ASub` accounts for generic consumers when their surrounding context is not empty ($\Sigma \neq \square$). We first infer their types with the empty context and put the type A into a new *matching* judgment $\Gamma \vdash A \approx \Sigma$ that matches the type A with Σ. Subsumption does not deal with applications and lambda expressions, since the rules that cover those expressions already deal with the cases when the context is not empty.

The matching judgment checks whether the type inferred in the subsumption premise matches up with the surrounding context. The rules for this judgment are defined by the structure of contexts. SubEmpty states that an empty context matches any type A. SubType states that a full type in a context matches type A when they are the same (syntactic) type. The most interesting rule is SubTerm, which matches a function type with an application argument followed by a context: e can be typed when the argument e has the input type A, and the output type B matches the context Σ. SubTerm shows that our typing and matching relations are *mutually dependent*, which adds some complexity to the algorithmic metatheory: related properties often need to be mutually proved. Interestingly, the typing premises for arguments in DApp1, and DApp2 correspond, respectively, to the typing premise of SubTerm and the first typing premise of ALam2. This observation is key to establishing the equivalence between the QTAS and the algorithmic system.

Metatheory. Algorithmic typing has the following properties.

Lemma 4.1 (Typing Implies Matching). *If* $\Gamma \vdash \Sigma \Rightarrow e \Rightarrow A$, *then* $\Gamma \vdash A \approx \Sigma$.

This property guarantees that all our typing rules have an invariant that the inferred type matches the context. This gives rise to the following corollary.

Corollary 4.2 (A full type context infers the same type). *If* $\Gamma \vdash A \Rightarrow e \Rightarrow B$, *then* $A = B$.

ASub has several restrictions to avoid overlapping, but a general form of subsumption that works for any expression is derivable. This lemma is proved with a generalization, that is proved together with Lemma 4.1 by induction on typing.

Lemma 4.3 (General Subsumption). *If* $\Gamma \vdash \square \Rightarrow e \Rightarrow A$ *and* $\Gamma \vdash A \approx \Sigma$, *then* $\Gamma \vdash \Sigma \Rightarrow e \Rightarrow A$.

Lemma 4.4 (Decidability of Matching). $\Gamma \vdash A \approx \Sigma$ *is decidable.*

Lemma 4.5 (Decidability of Typing). $\Gamma \vdash \Sigma \Rightarrow e \Rightarrow A$ *is decidable.*

The decidability of matching and typing is proved simultaneously. The measure used for typing is $\mathtt{size}(\Sigma) + \mathtt{size}(e)$, and the measure for matching is $\mathtt{size}(\Sigma)$. Note that the size of context Σ is defined in terms of the size of expressions. The definition of $\mathtt{size}(\Sigma)$ is:

$$\mathtt{size}(\square) = 0 \quad \mathtt{size}(A) = 0 \quad \mathtt{size}(\boxed{e} \mapsto \Sigma) = 1 + \mathtt{size}(e) + \mathtt{size}(\Sigma)$$

and the definition of $\mathtt{size}(e)$ is:

$$\mathtt{size}(i) = 1 \quad \mathtt{size}(x) = 1 \quad \mathtt{size}(\lambda x.\, e) = 1 + \mathtt{size}(e)$$
$$\mathtt{size}(e_1\ e_2) = 2 + \mathtt{size}(e_1) + \mathtt{size}(e_2) \quad \mathtt{size}(e : A) = 1 + \mathtt{size}(e)$$

4.4 Soundness and Completeness to the QTAS

Algorithmic typing is equivalent (sound and complete) to the corresponding QTAS in Fig. 5.

Soundness. Informally, the approach to achieve soundness is to extract all arguments ($\bar{e}$) from the context Σ and apply them back to e. After building an application expression $e\ \bar{e}$, we then assert that this constructed expression is well-typed in the QTAS. The general lemma that is needed to prove soundness is shown in the appendix. For space reasons, here we just show key results, which are corollaries of the general soundness lemma.

Corollary 4.6 (Soundness of Typing).

- *If* $\Gamma \vdash \square \Rightarrow e \Rightarrow A$, *then* $\Gamma \vdash_0 e : A$.
- *If* $\Gamma \vdash A \Rightarrow e \Rightarrow B$, *then* $\Gamma \vdash_\infty e : A$.

Types	$A, B, C, D ::= \texttt{Int} \mid \texttt{Float} \mid \texttt{Top} \mid A \to B \mid A \,\&\, B \mid \{a : A\}$
Constants	$c ::= i \mid u \mid + \mid +_i \mid +_u$
Expressions	$e ::= c \mid x \mid \lambda x.\, e \mid e_1\ e_2 \mid e : A \mid \{\overline{a = e}\} \mid e.a$
Environments	$\Gamma ::= . \mid \Gamma, x : A$
Check Counters	$m ::= 0 \mid \infty \mid S_c\ m \mid S_a\ m$
Counters	$n ::= m \mid S_i\ n$
Generic Consumers	$g ::= c \mid x \mid e : A \mid \{\overline{a = e}\}$
Surrounding Contexts	$\Sigma ::= \square \mid A \mid \boxed{e} \mapsto \Sigma \mid \boxed{a} \mapsto \Sigma$

Fig. 10. Syntax of $C_{\&}$.

Completeness. Compared with soundness, completeness is easier to prove. We also need an auxiliary relation that relates counters and a context Σ. We only show the key results here in the form of corollaries. The generalized completeness theorem can be found in the appendix.

Corollary 4.7 (Completeness of Typing).

- *If* $\Gamma \vdash_0 e : A$, *then* $\Gamma \vdash \square \Rightarrow e \Rightarrow A$.
- *If* $\Gamma \vdash_\infty e : A$, *then* $\Gamma \vdash A \Rightarrow e \Rightarrow A$.

5 A Calculus with Intersection Types, Overloading and Records

In this section, we present the contextual intersection calculus $C_{\&}$. $C_{\&}$ features subtyping, intersection types, a simple form of overloading, and records. The QTAS and algorithmic typing of $C_{\&}$ is formalized, and their soundness and completeness are proved. The main purpose of $C_{\&}$ is to show how contextual typing can scale up to more complex calculi, and deal with subtyping. We also illustrate that record labels can be another form of contextual information, helpful to aid with type inference of record projections. Additionally, we show that, with more variety of contextual information, counters are enriched to *qualify* the different kinds of contextual information.

5.1 Syntax

Types and Expressions. The syntax of $C_{\&}$ is shown in Fig. 10. Four new types are introduced into $C_{\&}$: a `Float` type; a `Top` type, which is the supertype of all types; intersection types $A \,\&\, B$; and single field records $\{a : A\}$. Multi-field record types are encoded using top, intersection and single field record types [Reynolds 1997]:

$$\{\} \triangleq \texttt{Top} \qquad \{a_1 : A_1, \ldots, a_i : A_i\} \triangleq \{a_1 : A_1\} \,\&\, \ldots \,\&\, \{a_i : A_i\}$$

In expressions, we separate out all constants into a new category: i is an integer and u is a floating point number. The overloaded addition $+$ can work on integers and floats. Furthermore, $+_i$ and $+_u$ are two specialized partially applied additions that work only on integers and floats respectively. Two more new constructs are records, which are constructed by a sequence of labels a bound to expressions e, and record projection $e.a$. We define a metafunction $[\![c]\!]$ that maps constants to types. Note that we use I and F as abbreviations for `Int` and `Float`, respectively.

$$[\![i]\!] = I \quad [\![u]\!] = F \quad [\![+]\!] = (I \to I \to I) \,\&\, (F \to F \to F) \quad [\![+_i]\!] = I \to I \quad [\![+_u]\!] = F \to F$$

Qualifying counters. To account for the richer form of contextual information in the calculus, we use a richer formulation of counters. Instead of a single successor used in the counters in Section 3, we have three, more refined, forms of successors: S_i, S_c and S_a. S_i counts the availability of *inferable* argument types (and the uses of `CDApp2`), and is the closest to the successor used in Section 3.4. The successor counter (S_a) counts labels, which are helpful to aid with the inference of types for projections and provides a distinct use of counters. In addition, we introduce an extra counter S_c that counts the number of arguments that check against a type (and the uses of `CDApp1`). Tracking

checkable arguments is important because *subtyping* exploits information about arguments that can be checked: the subtyping derivations will depend on the arguments in the context.

One useful observation here is that there is a mismatch between the syntax of counters and contexts in the STLC calculus. The QTAS counts the needed contextual information for inferable arguments; while contexts store all possible arguments, whether they are inferable or not. For our STLC calculus this mismatch does not matter, so we do not need to track checkable arguments. In $C_\&$ we need to be more precise and track checkable arguments, and there is a closer correspondence between counters and contexts.

Note that in $C_\&$ a well-formed counter is wrapped around check counters. This exposes an interesting structure in applications with multiple arguments $f\ e_1\ e_2\ \ldots\ e_n$. The inference result of inner arguments can assist the inference of function f's type, and checking outer arguments. In other words, the general structure of an application is of the form $f\ \overline{i}\ \overline{c}$ where $\overline{i}$ denotes a sequence of inferable arguments, and $\overline{c}$ denotes a sequence of checkable arguments. For example, $(\lambda x.\ ((\lambda f.\ f\ x) : (I \to I) \to I))\ 1\ (\lambda x.\ x)$ type checks, since the first argument 1 is inferable and helps inferring the type for the corresponding lambda application consumer. Then the second argument $(\lambda x.\ x)$ can be checked with $I \to I$. However, if we swap the two arguments, $(\lambda f.\ ((\lambda x.\ f\ x) : (I \to I)))\ (\lambda x.\ x)\ 1$ does not type-check since $\lambda x.\ x$ is not inferable and cannot help inferring the corresponding lambda application consumer. Note also, that all inferable arguments are checkable. Therefore an application such as $f\ 1\ (\lambda x.\ x)\ 2$ can be interpreted as being of the form $f\ i\ c\ c$. Any $f\ \overline{i}\ \overline{c}\ \overline{i}$ expressions are also $f\ \overline{i}\ \overline{c}\ \overline{c}$ expressions: we can always check inferable arguments. Once we encounter an argument that must be checked, then all subsequent arguments can be checked. We need to apply `(C)DApp1`, which provides enough information to check all subsequent arguments. Interestingly, similar structures appear in applications for approaches based on *local type inference* [Pierce and Turner 2000].

To conclude, in $C_\&$ counters have a broader role, and they do not only quantify, but they also qualify available contextual type information. In $C_\&$ counters can be interpreted as a kind of mask that classifies what each piece of contextual information in the surrounding context is.

Surrounding Contexts. Compared with the changes in counters, there is only one small change in contexts. We allow labels in contexts: $\boxed{a} \mapsto \Sigma$ is a label followed by a context. In $C_\&$ there is a new elimination form: record projections $e.a$. Thus, as described in Section 4.3, we need to identify the information from the elimination form that is helpful for the corresponding consumers, which includes record expressions (the introduction form), as well as variables and annotated expressions. For projections, labels are the information that is needed to help the consumers establish typeability. Thus labels are now also tracked in the context. Note also that in $C_\&$ we now achieve a close match between counters and contexts. For example, $S_i\ S_c\ S_a\ 0$ matches $\boxed{e_1} \mapsto \boxed{e_2} \mapsto \boxed{a_1} \mapsto \square$, where e_1 infers and e_2 checks during typing.

5.2 QTAS: Typing and Subtyping

Typing. The QTAS for $C_\&$ is shown at the top of Fig. 11. The $n-$ meta-operation is the same as the first calculus. `CDApp1` and `CDApp2` account for the two application rules. Rule `CDApp1` tracks one more counter. As before, we check whether e_2 has the type A with full contextual information. Differently to previous QTASs, function e_1 has the arrow type $A \to B$ with the incremented counter $S_c\ m$ to denote the availability of one additional checkable argument. This information is helpful for subsumption. In the `CDLam` rule it is important to note that the decrement function is only defined for ∞ and $S_i\ n$, since the type of the argument must be known.

`CDRcd` deals with the introduction of records and requires some explanation. One may expect that expressions like $\{a = \lambda x.\ x\}.a\ 1$ are typeable, by propagating the surrounding context (with 1), into the lambda. However, consider $\{a_1 = 1, a_2 = \lambda x.\ x\}.a_1$. We expect that this expression has

$\boxed{\Gamma \vdash_n e : A}$ (Under Γ, expression e needs n contextual information to have type A.)

$$\textsf{CDConst}\ \frac{}{\Gamma \vdash_0 c : [\![c]\!]} \qquad \textsf{CDVar}\ \frac{x : A \in \Gamma}{\Gamma \vdash_0 x : A} \qquad \textsf{CDAnn}\ \frac{\Gamma \vdash_\infty e : A}{\Gamma \vdash_0 (e : A) : A} \qquad \textsf{CDLam}\ \frac{\Gamma, x : A \vdash_{n-} e : B}{\Gamma \vdash_n \lambda x.\, e : A \to B}$$

$$\textsf{CDApp1}\ \frac{\Gamma \vdash_{(S_c\ m)} e_1 : A \to B \qquad \Gamma \vdash_\infty e_2 : A}{\Gamma \vdash_m e_1\ e_2 : B} \qquad \textsf{CDApp2}\ \frac{\Gamma \vdash_{(S_i\ n)} e_1 : A \to B \qquad \Gamma \vdash_0 e_2 : A}{\Gamma \vdash_n e_1\ e_2 : B}$$

$$\textsf{CDSub}\ \frac{\Gamma \vdash_0 e : A \qquad n \neq 0 \qquad A <:_n B}{\Gamma \vdash_n e : B} \qquad \textsf{CDRcd}\ \frac{\overline{\Gamma \vdash_0 e_i : A_i}}{\Gamma \vdash_0 \overline{\{a_i = e_i\}} : \overline{\{a_i : A_i\}}} \qquad \textsf{CDPrj}\ \frac{\Gamma \vdash_{(S_a\ m)} e : \{a : A\}}{\Gamma \vdash_m e.a : A}$$

$\boxed{A <:_n B}$ (A is a subtype of B and n contextual information is known for B.)

$$\textsf{CDSRefl}\ \frac{}{A <:_0 A} \qquad \textsf{CDSInt}\ \frac{}{\texttt{Int} <:_\infty \texttt{Int}} \qquad \textsf{CDSFloat}\ \frac{}{\texttt{Float} <:_\infty \texttt{Float}} \qquad \textsf{CDSTop}\ \frac{}{A <:_\infty \texttt{Top}} \qquad \textsf{CDSArr}\ \frac{C <:_\infty A \qquad B <:_{n\ominus} D}{A \to B <:_n C \to D}$$

$$\textsf{CDSRcd}\ \frac{A <:_{m\ominus} B}{\{a : A\} <:_m \{a : B\}} \qquad \textsf{CDSAndL}\ \frac{A <:_n C \qquad n \neq 0}{A\ \&\ B <:_n C} \qquad \textsf{CDSAndR}\ \frac{B <:_n C \qquad n \neq 0}{A\ \&\ B <:_n C} \qquad \textsf{CDSAnd}\ \frac{A <:_\infty B \qquad A <:_\infty C}{A <:_\infty B\ \&\ C}$$

Fig. 11. QTAS typing and subtyping for $C_\&$.

type `Int`, but we cannot type a_2 under any surrounding context: it is not possible to propagate contextual information to a_2. A similar situation happens with $\{a_1 = 1, a_2 = \lambda x.\, x\} : \{a_1 : \texttt{Int}\}$, where a record expression may be checked against a type that has *fewer fields* than the record expression itself. Both of these bring up the question of what to do for the expressions in the fields for which there is no contextual information (such as a_2). To avoid this problem we require that all fields must have inferable types in `CDRcd`. An alternative approach could be to employ a mixed strategy where the fields that are present in the type/context are checked, and the fields that are missing require inference. But this approach is more complex (although it can be algorithmically modeled) and seems ad-hoc. Thus we opt for the simpler approach of requiring all record fields to have inferred types.

For projections (rule `CDPrj`), we take the same strategy as `App1`. We increment the counter and delegate the job of label selection to subsumption. Thus, our typing relation *does not* require auxiliary relations for dealing with projections and applications. In the subsumption rule `CDSub`, we introduce subtyping. Interestingly, subtyping is also parametrized by a counter, enabling it to deal with contextual type information.

To illustrate how typing works, consider an example with overloaded addition. We omit the derivation of subtyping, which will be detailed later.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash_0 + : (I \to I \to I)\ \&\ (F \to F \to F)}\ \textsf{CDConst} \quad \ldots}{\Gamma \vdash_{(S_c\ S_c\ 0)} + : I \to I \to I}\ \textsf{CDSub} \qquad \Gamma \vdash_\infty 1 : I}{\Gamma \vdash_{(S_c\ 0)} + 1 : I \to I}\ \textsf{CDApp1} \qquad \Gamma \vdash_\infty 2 : I}{\Gamma \vdash_0 + 1\ 2 : I}\ \textsf{CDApp1}$$

When applying + to two integers, the counter will be incremented twice by the successor S_c, which tells that two arguments are needed. With additional contextual information, the type of + can be

upcast to a function type via the subsumption rule which, in this case, is $I \to I \to I$. Projections are dealt with similarly. For example, assuming $r : \{x : \texttt{Int}, y : \texttt{Float}\}$, then $r.x$ infers `Int`.

Subtyping. Unlike conventional subtyping relations, our QTAS subtyping, shown at the bottom of Fig. 11, also carries a counter. The counter n expresses how much contextual information is known for B, whereas A is fully known. `CDSRefl` describes the situation when the counter is 0, which means that we have no information for B. In this case we pick type A for the supertype. If the counter is ∞, then we know everything about type B, corresponding to traditional subtyping. `CDSInt` and `CDSFloat` are unsurprising and describe reflexivity for the two base types. `CDSTop` is also standard. The rule `CDSArr` is interesting since it can deal with partially known information about B. Note that n cannot be 0, since it needs to be decreasable:

$$(S_i\ n)\ominus = n \quad \infty\ominus = \infty \quad (S_c\ m)\ominus = m \quad (S_a\ m)\ominus = m$$

Thus C is known. In the first premise, since both A and C are known, the counter is set to ∞. For the second premise, which performs subtyping for the output types, we decrease the counter. `CDSRcd` works similarly to `CDSArr`: the label information must be known for the supertype, although the type of the field may or may not be known. The rules `CDSAndL` and `CDSAndR` are standard for intersection types if we ignore counter information. The counters express that we can have all or just partial information for C. Interestingly, for `CDSAnd`, all the information for the supertype must be known. Thus, the rule is just the standard rule when an intersection appears in the supertype. We continue our example, to illustrate how the derivation of subtyping proceeds in subsumption.

$$\dfrac{\dfrac{\dfrac{}{I <:_{\infty} I}\ \texttt{CDSInt} \qquad \dfrac{\dfrac{}{I <:_{\infty} I}\ \texttt{CDSInt} \qquad \dfrac{}{I <:_{0} I}\ \texttt{CDSRefl}}{(I \to I) <:_{(S_c\ 0)} (I \to I)}\ \texttt{CDSArr}}{(I \to I \to I) <:_{(S_c\ S_c\ 0)} (I \to I \to I)}\ \texttt{CDSArr}}{(I \to I \to I)\ \&\ (F \to F \to F) <:_{(S_c\ S_c\ 0)} (I \to I \to I)}\ \texttt{CDSAndL}$$

Metatheory. QTAS subtyping has following properties:

Lemma 5.1 (Reflexivity of Subtyping (Infinity)). *$A <:_{\infty} A$*

Lemma 5.2 (Transitivity of Subtyping). *If $A <:_n B$ and $B <:_n C$, then $A <:_n C$.*

5.3 Algorithmic Typing and Subtyping

Algorithmic typing. We present the rules of algorithmic typing at the top of Fig. 12. Many rules are the same as the type system in Section 4.3. For records, similarly to the QTAS in Fig. 11, `CARcd` deals with the introduction of records. For a record projection $e.a$ (rule `CAPrj`), we push the label a into the context and infer the expression e with the single field record type $\{a : A\}$. `CASub` is interesting since subtyping needs to deal with contextual information. Unlike conventional subtyping, which compares two types, our contextual subtyping relation compares a type with a surrounding context, which has partial information about the supertype. While comparing the type with the context, we compute a supertype that matches the information in the context.

Algorithmic contextual subtyping. Algorithmic subtyping is presented at the bottom of Fig. 12. Γ, A and Σ are taken as inputs and B is an output. We group `CASInt`, `CASTop`, `CASArr`, `CASRcd` and `CASAnd` together, since they cover the same situation where contexts are fully known types. In those cases the computed supertype is exactly the type in the context. Rule `CASArr` is standard, except for the additional computed types. The computed types in the premises are ignored since the full supertype is already known. `CASAnd` and `CASRcd` are also quite standard and work similarly to `CASArr`. In `CASEmpty` the computed supertype is A. In `CASTerm`, an arrow type is the subtype of a

$\boxed{\Gamma \vdash \Sigma \Rightarrow e \Rightarrow A}$ (Under environment Γ and context Σ, expression e infers type A.)

$$\frac{}{\Gamma \vdash \square \Rightarrow c \Rightarrow [\![c]\!]}\ \textsc{CAConst} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash \square \Rightarrow x \Rightarrow A}\ \textsc{CAVar} \qquad \frac{\Gamma \vdash \boxed{a} \mapsto \Sigma \Rightarrow e \Rightarrow \{a : A\}}{\Gamma \vdash \Sigma \Rightarrow e.a \Rightarrow A}\ \textsc{CAPrj}$$

$$\frac{\Gamma \vdash \boxed{e_2} \mapsto \Sigma \Rightarrow e_1 \Rightarrow A \to B}{\Gamma \vdash \Sigma \Rightarrow e_1\ e_2 \Rightarrow B}\ \textsc{CAApp} \qquad \frac{\Gamma, x : A \vdash B \Rightarrow e \Rightarrow C}{\Gamma \vdash A \to B \Rightarrow \lambda x.\ e \Rightarrow A \to C}\ \textsc{CALam1}$$

$$\frac{\overline{\Gamma \vdash \square \Rightarrow e_i \Rightarrow A_i}}{\Gamma \vdash \square \Rightarrow \overline{\{a_i = e_i\}} \Rightarrow \overline{\{a_i : A_i\}}}\ \textsc{CARcd} \qquad \frac{\Gamma \vdash \square \Rightarrow e_2 \Rightarrow A \qquad \Gamma, x : A \vdash \Sigma \Rightarrow e \Rightarrow B}{\Gamma \vdash \boxed{e_2} \mapsto \Sigma \Rightarrow \lambda x.\ e \Rightarrow A \to B}\ \textsc{CALam2}$$

$$\frac{\Gamma \vdash A \Rightarrow e \Rightarrow B}{\Gamma \vdash \square \Rightarrow e : A \Rightarrow A}\ \textsc{CAAnn} \qquad \frac{\Gamma \vdash \square \Rightarrow g \Rightarrow A \qquad \Sigma \neq \square \qquad \Gamma \vdash A <: \Sigma \rightsquigarrow B}{\Gamma \vdash \Sigma \Rightarrow g \Rightarrow B}\ \textsc{CASub}$$

$\boxed{\Gamma \vdash A <: \Sigma \rightsquigarrow B}$ (Under environment Γ, A matches the context Σ and is a subtype of B.)

$$\frac{}{\Gamma \vdash \mathtt{Int} <: \mathtt{Int} \rightsquigarrow \mathtt{Int}}\ \textsc{CASInt} \qquad \frac{}{\Gamma \vdash A <: \mathtt{Top} \rightsquigarrow \mathtt{Top}}\ \textsc{CASTop} \qquad \frac{}{\Gamma \vdash A <: \square \rightsquigarrow A}\ \textsc{CASEmpty}$$

$$\frac{}{\Gamma \vdash \mathtt{Float} <: \mathtt{Float} \rightsquigarrow \mathtt{Float}}\ \textsc{CASFloat} \qquad \frac{\Gamma \vdash C <: A \rightsquigarrow A' \qquad \Gamma \vdash B <: D \rightsquigarrow D'}{\Gamma \vdash A \to B <: C \to D \rightsquigarrow C \to D}\ \textsc{CASArr}$$

$$\frac{\Gamma \vdash A \Rightarrow e \Rightarrow C \qquad \Gamma \vdash B <: \Sigma \rightsquigarrow D}{\Gamma \vdash A \to B <: \boxed{e} \mapsto \Sigma \rightsquigarrow A \to D}\ \textsc{CASTerm} \qquad \frac{\Gamma \vdash A <: B \rightsquigarrow B'}{\Gamma \vdash \{a : A\} <: \{a : B\} \rightsquigarrow \{a : B\}}\ \textsc{CASRcd}$$

$$\frac{\Gamma \vdash A <: \Sigma \rightsquigarrow C}{\Gamma \vdash \{a : A\} <: \boxed{a} \mapsto \Sigma \rightsquigarrow \{a : C\}}\ \textsc{CASLabel} \qquad \frac{\Gamma \vdash A <: \Sigma \rightsquigarrow C \qquad \Sigma \neq \square}{\Gamma \vdash A \,\&\, B <: \Sigma \rightsquigarrow C}\ \textsc{CASAndL}$$

$$\frac{\Gamma \vdash B <: \Sigma \rightsquigarrow C \qquad \Sigma \neq \square}{\Gamma \vdash A \,\&\, B <: \Sigma \rightsquigarrow C}\ \textsc{CASAndR} \qquad \frac{\Gamma \vdash A <: B \rightsquigarrow B' \qquad \Gamma \vdash A <: C \rightsquigarrow C'}{\Gamma \vdash A <: B \,\&\, C \rightsquigarrow B \,\&\, C}\ \textsc{CASAnd}$$

Fig. 12. Algorithmic typing and subtyping for $C_{\&}$.

term e followed by a context Σ. We use the type A as the context for typing e, and check subtyping between B and Σ inductively, while computing a supertype D for B. The computed supertype is then $A \to D$. CASLabel works similarly: a record type is a subtype of a label a followed by a context if their labels are the same. CASAndL and SAndR describe intersection cases. If $A \,\&\, B$ is a subtype of the context Σ, then either A or B are subtypes of Σ resulting in a supertype C. Note that backtracking is needed here: we have to try with both CASAndL and CASAndR. This is expected since this form of backtracking is inherent to subtyping relations with intersection types.

Metatheory. Algorithmic typing and subtyping have the following properties.

Lemma 5.3 (A full type context infers the same type). *If* $\Gamma \vdash B \Rightarrow e \Rightarrow A$, *then* $A = B$.

Lemma 5.4 (A full type context computes the same type). *If* $\Gamma \vdash A <: B \rightsquigarrow C$, *then* $B = C$.

Lemma 5.5 (Typing implies Subtyping). *If* $\Gamma \vdash \Sigma \Rightarrow e \Rightarrow A$, *then* $\Gamma \vdash A <: \Sigma \rightsquigarrow A$.

Lemma 5.6 (General Subsumption of Algorithmic Typing). *If* $\Gamma \vdash \square \Rightarrow e \Rightarrow A$ *and* $\Gamma \vdash A <: \Sigma \rightsquigarrow A'$, *then* $\Gamma \vdash \Sigma \Rightarrow e \Rightarrow A'$.

5.4 Key Properties

The metatheory of $C_{\&}$ requires significant changes, compared to the STLC calculus, because of subtyping. Nonetheless soundness and completeness proofs employ similar strategies as the STLC

calculus, with auxiliary relations relating counters to contexts. Furthermore the key corollaries are still essentially of the same form.

Corollary 5.7 (Soundness of Typing).

- *If* $\Gamma \vdash \square \Rightarrow e \Rightarrow A$, *then* $\Gamma \vdash_0 e : A$.
- *If* $\Gamma \vdash A \Rightarrow e \Rightarrow B$, *then* $\Gamma \vdash_\infty e : A$.

Corollary 5.8 (Completeness of Typing).

- *If* $\Gamma \vdash_0 e : A$, *then* $\Gamma \vdash \square \Rightarrow e \Rightarrow A$.
- *If* $\Gamma \vdash_\infty e : A$, *then* $\Gamma \vdash A \Rightarrow e \Rightarrow A$.

Annotatability. One point worth mentioning about annotatability is that, for records, if the TAS expression e in a record field $\{a = e\}$ needs contextual information, then the QTAS expression must be annotated with its type. This is needed because records require all the fields to have inferred types. Except for this, the annotatability guidelines remain the same as in the STLC. Note that the rules for the TAS can be recovered by: 1) erasing the counter information in the rules in Fig. 11; and 2) ommiting the `CDAnn` rule. The full elaboration rules from the TAS to the QTAS, and the extended `need` function can be found in the appendix.

Theorem 5.9 (Strong Annotatability).

(1) If $\Gamma \vdash e : A \rightsquigarrow e'$, *then* $\Gamma \vdash_{(\mathsf{need}\ e)} e' : A$. *(2) If* $\Gamma \vdash e : A \rightsquigarrow e'$, *then* $\Gamma \vdash_0 (e' : A) : A$.

Type safety. The type-safety of $C_\&$ is proved by first showing that typing is preserved between the QTAS and the TAS. We use a simple erasure function $|e|$ that erases all type annotations, so that we obtain an unannotated expression that is typeable in the TAS. Then we prove that the TAS is type sound via standard preservation and progress theorems. The small-step operational semantics $e \hookrightarrow e'$ is shown in the appendix.

Theorem 5.10 (Type preservation after erasure to TAS). *If* $\Gamma \vdash_n e : A$, *then* $\Gamma \vdash |e| : A$.

Theorem 5.11 (Preservation of TAS). *For well-formed e, if* $\cdot \vdash e : A$ *and* $e \hookrightarrow e'$, *then* $\cdot \vdash e' : A$.

Theorem 5.12 (Progress of TAS). *For well-formed e, if* $\cdot \vdash e : A$, *then* $\exists\, e', e \hookrightarrow e'$ *or e is a value.*

Determinism and decidability. Typing and subtyping are non-deterministic, without further restrictions, due to the presence of the subtyping rules for intersection types, which are overlapping. To recover determinism in algorithmic typing, we need to impose some additional well-formedness conditions on environments Γ, surrounding contexts Σ, types A and expressions e, to deal with non-determinism caused by intersections. Well-formed intersections can either contain: record types with distinct labels, or function types where the domains are distinct primitive types (`Int` and `Float`). The latter restriction is sufficient to cover the form of overloading that we have in calculus, but it is rather strict. We leave for future work relaxing the restrictions to allow more forms of functional intersections and overloading. Under this set of restrictions we can prove determinism:

Lemma 5.13 (Determinism of Typing). *For well-formed* Γ, Σ *and e, if* $\Gamma \vdash \Sigma \Rightarrow e \Rightarrow A$ *and* $\Gamma \vdash \Sigma \Rightarrow e \Rightarrow B$, *then* $A = B$.

Lemma 5.14 (Determinism of Subtyping). *For well-formed* Γ, Σ *and A, if* $\Gamma \vdash A <: \Sigma \rightsquigarrow B$ *and* $\Gamma \vdash A <: \Sigma \rightsquigarrow C$, *then* $B = C$.

Furthermore, using the same restrictions we can prove the decidability of typing and subtyping:

Theorem 5.15 (Decidability of Typing). *For well-formed* Γ, Σ *and e,* $\Gamma \vdash \Sigma \Rightarrow e \Rightarrow A$ *is decidable.*

Theorem 5.16 (Decidability of Subtyping). *For well-formed* Γ, Σ *and A,* $\Gamma \vdash A <: \Sigma \rightsquigarrow B$ *is decidable.*

The measure used for decidability of $C_{\&}$ is similar to Section 4.3. With soundness, completeness and decidability of algorithmic formulation, we can obtain the decidability of QTAS as a corollary.

Corollary 5.17 (Decidability of the QTAS). *For well-formed* Γ *and* e, $\Gamma \vdash_0 e : A$ *is decidable.*

6 Related Work

In Section 2 we have already covered the closest related work: standard bidirectional typing [Dunfield and Krishnaswami 2022], and the let arguments go first approach [Xie and Oliveira 2018]. As discussed in Section 3, contextual typing can be viewed as a generalization of both approaches. In this section we focus on discussing other closely related work. Other variants of bidirectional typing, which are further away from our work, are covered in Dunfield and Krishnaswami's survey.

Forms of contextual typing. Xie and Oliveira's application stack Ψ is a form of context. As we have seen our notion of surrounding context subsumes the application stack. There are several other works [Bierman et al. 2014; Polikarpova et al. 2016; Pottier and Régis-Gianas 2006] that employ typing relations similar to our algorithmic typing. Typically the form of the typing relation, adapted to our notation here, is a variant of $\Gamma \vdash A \Rightarrow e \Rightarrow B$, where the context is a *type* A. Bierman et al. model the type inference approach adopted by TypeScript. Their *expression contextual typing judgment* fills the role of the checking mode. The judgment is used to type-check n-ary uncurried applications and it is useful to instantiate type variables coming from the function, and to type-check the arguments of the function. Polikarpova et al.'s contextual type is an unrefined type, which gets refined as more information is learned from doing inference on the subterms. Pottier and Régis-Gianas use a context that is a *shape*. A shape has the form $\overline{\alpha}.A$, where $\overline{\alpha}$ denotes a collection of *flexible* type variables that are bound within A. A flexible type variable is similar to a unification variable and can match with other types. In comparison to our work, we allow more forms of contextual type information, since we can also have term arguments or labels. Moreover, those works do not have a direct correspondence to the rule `DApp2` in our work.

Norell [2007] formalizes Agda's typing. Typing has an implicit context tracking a constraint set that is used to aid type inference. This constraint set can also be viewed as a form of contextual typing, and is helpful to defer type checking tasks until more information is discovered. However, the constraint set is a global form of context that is updated and threaded through typing derivations.

Bidirectional typing with subtyping. Bidirectional typing has been extensively applied to type systems with subtyping. Several bidirectional type systems with intersection types [Davies and Pfenning 2000; Huang et al. 2021; Xue et al. 2022] require changes to the application rule to allow functions to infer intersection types. An auxiliary relation is used to upcast the intersection type to a function type. Some of these works [Huang et al. 2021; Xue et al. 2022] also support distributivity rules for subtyping and model records via intersection types, where a similar problem appears for projections. In a projection $e.a$ the type of e can be an intersection of single field records. Then we need an auxiliary relation that takes the intersection and the label and selects the corresponding field type. Our contextual typing approach avoids changes to existing rules and specialized auxiliary relations. Instead, subsumption and contextual subtyping deal with intersection types in a single place. Rioux et al. [2023] also have a calculus with intersection types, but additionally support union types and more distributivity rules for subtyping. They need a special application rule with a sophisticated auxiliary relation to deal with types of functions that can be intersection or union types. This relation requires both the type of the function, as well as the type of the argument. So the application rule also requires the inference of the argument, and is a variant of the `App3` rule described in Section 2.1. Consequently, their rule is, in some cases, weaker than `DApp1` and there is

no correspondence to the DApp2 rule either. While our work does not cover all the features studied by Rioux et al., we believe that it is possible to scale up our techniques to deal with those features.

In bidirectional type systems with polymorphism [Dunfield and Krishnaswami 2013; Zhao et al. 2019], specialized auxiliary relations are also needed for applications. In this case, a function can have a polymorphic type, which needs to be instantiated to become a function type. Then we can use the domain of the instantiated function type to check the argument of the function. We believe that our approach can also be used in type systems with polymorphism, and avoid the need for specialized rules and relations. However, those type systems introduce important complications. In particular, we need to also employ a form of unification variables to aid with instantiation. Studying the interaction between contextual typing and polymorphism is an interesting line for future work.

Spines. Application spines [Cervesato and Pfenning 2003] model the arguments of n-ary applications of the form $h\ e_1 \ldots e_n$. Cervesato and Pfenning propose a *syntactic* representation of spines, motivated by gaining efficient access to the head of an application (h). The notions of a surrounding context and application consumer in our work are closely related to the concept of an application spine and its head, respectively. However, the structure of a surrounding context is more general than a spine, and the head of an application is not the same as the application consumer. For an application $e_1\ e_2$ what we are looking for is the *application consumer* for e_2. In some cases, such as $x\ 1\ 2$, the concept of an application head coincides with the application consumer. However, consider the application $(\lambda x.\ y)\ 1\ (\lambda z.\ z)$ in Section 4.1. Here y is a variable, with the type $(I \to I) \to I$ in the typing context. In this case y would be the application consumer that determines which application rule to use to type check the (outer) application. But this example illustrates that the application consumer does not always coincide with the application head, which is $\lambda x.\ y$ in this case. In addition to tracking information about applied arguments, surrounding contexts also track known type information, which are not tracked in syntactic representations of spines.

The notion of spines has been found to be useful in type inference for languages with implicit polymorphism [Jenkins and Stump 2018; Leijen 2008; Mercer et al. 2022; Serrano et al. 2020, 2018]. Most of these works [Leijen 2008; Mercer et al. 2022; Serrano et al. 2020, 2018], employ a syntactic representation of spines (also called argument lists), similar to the representation proposed by Cervesato and Pfenning, to assist type inference. *Spine-local type inference* [Jenkins and Stump 2018] also employs the concept of spines, but without requiring a syntactic representation for spines in the syntax for terms. Instead they use standard lambda calculus syntax, like in our own work. Their typing relation employs bidirectional typing. Their rule for applications employs some auxiliary relations that are responsible for finding the head of an application, and collect arguments of an application spine. In addition, they also track known type information about the output type of an application, when available. Then their typing relation is able to accept:

$$\texttt{pair} : \forall X\ Y.\ X \to Y \to \langle X \times Y \rangle \vdash \texttt{pair}\ (\lambda x.\ x)\ 1 \Leftarrow \langle (\texttt{Int} \to \texttt{Int}) \times \texttt{Int} \rangle$$

Here we want to find the implicit instantiations for X and Y and deal with the first argument $\lambda x.\ x$. Because they find the type of the head ($\forall X\ Y.\ X \to Y \to \langle X \times Y \rangle$) and they track the output type of the application ($\langle (\texttt{Int} \to \texttt{Int}) \times \texttt{Int} \rangle$), when they find the argument $\lambda x.\ x$ they are able to conclude that $X = \texttt{Int} \to \texttt{Int}$, and check the lambda with that type. Unlike our work, spine-local type inference cannot type-check terms like $(\lambda x\ y.\ x + y)\ 1\ 2$, and it lacks the power afforded by the AppS rule [Xie and Oliveira 2018]. Although the output types of applications are partly tracked, spine-local type inference also rejects programs such as $((\lambda x.\ x)\ 1) : \texttt{Int}$, which are accepted using rule App2. So spine-local type inference does not provide the power afforded by App2 either.

Local type inference. Local type inference [Pierce and Turner 2000] combines bidirectional typing with local argument synthesis, to deal with implicit polymorphism. Instead of spines, local type inference uses n-ary *uncurried* applications, which also gives direct access to the head of an

application. This representation, combined with local argument synthesis, enables the instantiation of type arguments for applications. *Colored local type inference* [Odersky et al. 2001] is an extension of local type inference. Like us, Odersky et al. observe that a drawback of bidirectional typing is that it cannot exploit partial type information. To address this limitation they propose *colored types*, in which types are annotated with two propagation directions: *inherited*; or *synthesized*. An inherited type is a known type, and should be propagated downwards the AST. A synthesized type, on the other hand, must be propagated upwards the AST, after some information about the type is discovered. From a bidirectional typing perspective, checked types are types that are fully inherited, whereas inferred types are types that are fully synthesized. Colored types enable types where some portions are inherited and others synthesized. Nonetheless, this adds complexity in the syntax of types, since all types need to be annotated with propagation directions. Consequentely, the typing rules also become more complicated in order to deal with colored types. Contextual typing deals with partial type information via the surrounding context, and does not require changes to the syntax of types. Thus the propagation of partial type information is considerably simpler. However, we currently do not deal with implicit polymorphism, which is left to future work.

7 Conclusion

Contextual typing is a lightweight form of type inference that exploits partially known contextual information. It enables several improvements over bidirectional typing. Firstly, fewer annotations are needed, by having a more powerful treatment for applications, and propagating contextual information. This is achieved without backtracking, keeping the approach efficient. More powerful application rules also help with binding constructs and avoid duplicated rules. Secondly, annotatability becomes clearer, and specifications of typing describing where annotations are needed are made precise via a QTAS. Finally, for type systems with subtyping, contextual subsumption deals with partially known contextual information, and avoids changes in other rules and specialized auxiliary relations. For implementations, instead of two mutually recursive inference and checking functions, we need mutually recursive typing and subtyping/matching functions.

To design a type system with contextual typing, an informal design recipe is to start with the corresponding TAS. Then there are two main tasks: 1) determining contextual type information and the structure of the surrounding context; and 2) designing the QTAS and counters. For (1) we must analyse the elimination forms in the language and determine the information that needs to be tracked in the context to aid the consumers. For (2), the design of the QTAS and counters is somewhat interconnected. A simpler case is when there is only a single QTAS typing rule for each elimination form. In this case, the design of counters is simple: we simply need a different kind of successor for each entry in the surrounding context. The main complication that can show up when designing a QTAS and counters is that we may want multiple rules for the same elimination form. This is what happens with applications, where we wish to have 2 different rules for added flexibility. In the general case, we need a successor counter for each typing rule dealing with the elimination form, although for STLC it is possible to have a simpler design with only one successor.

We only cover a small set of features in this work. More study is needed to show the practical applicability of contextual typing to other features. The interaction between contextual typing and polymorphism is a particularly interesting direction for future work.

Acknowledgments

We are grateful to the anonymous reviewers for their valuable comments, and to Shengyi Jiang for his insight of substitution lemma and help in complex induction. This work has been sponsored by Hong Kong Research Grant Council project number 17209821.

References

Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. 2012. A Bi-Directional Refinement Algorithm for the Calculus of (Co)Inductive Constructions. *Log. Methods Comput. Sci.* 8, 1 (2012). https://doi.org/10.2168/LMCS-8(1:18)2012

Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. 1983. A Filter Lambda Model and the Completeness of Type Assignment. *J. Symb. Log.* 48, 4 (1983), 931–940. https://doi.org/10.2307/2273659

Gavin M. Bierman, Martín Abadi, and Mads Torgersen. 2014. Understanding TypeScript. In *ECOOP 2014 - Object-Oriented Programming - 28th European Conference, Uppsala, Sweden, July 28 - August 1, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8586)*, Richard E. Jones (Ed.). Springer, 257–281. https://doi.org/10.1007/978-3-662-44202-9_11

Iliano Cervesato and Frank Pfenning. 2003. A Linear Spine Calculus. *J. Log. Comput.* 13, 5 (2003), 639–688. https://doi.org/10.1093/LOGCOM/13.5.639

Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. 1981. Functional Characters of Solvable Terms. *Math. Log. Q.* 27, 2-6 (1981), 45–58. https://doi.org/10.1002/MALQ.19810270205

Thierry Coquand. 1996. An Algorithm for Type-Checking Dependent Types. *Sci. Comput. Program.* 26, 1-3 (1996), 167–177. https://doi.org/10.1016/0167-6423(95)00021-6

Rowan Davies and Frank Pfenning. 2000. Intersection types and computational effects. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, Martin Odersky and Philip Wadler (Eds.). ACM, 198–208. https://doi.org/10.1145/351240.351259

Jana Dunfield and Neel Krishnaswami. 2022. Bidirectional Typing. *ACM Comput. Surv.* 54, 5 (2022), 98:1–98:38. https://doi.org/10.1145/3450952

Jana Dunfield and Neelakantan R. Krishnaswami. 2013. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, Greg Morrisett and Tarmo Uustalu (Eds.). ACM, 429–442. https://doi.org/10.1145/2500365.2500582

Jana Dunfield and Frank Pfenning. 2004. Tridirectional typechecking. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, Neil D. Jones and Xavier Leroy (Eds.). ACM, 281–292. https://doi.org/10.1145/964001.964025

Xuejing Huang, Jinxu Zhao, and Bruno C. d. S. Oliveira. 2021. Taming the Merge Operator. *J. Funct. Program.* 31 (2021), e28. https://doi.org/10.1017/S0956796821000186

Christopher Jenkins and Aaron Stump. 2018. Spine-local Type Inference. In *Proceedings of the 30th Symposium on Implementation and Application of Functional Languages, IFL 2018, Lowell, MA, USA, September 5-7, 2018*, Matteo Cimini and Jay McCarthy (Eds.). ACM, 37–48. https://doi.org/10.1145/3310232.3310233

Daan Leijen. 2008. HMF: simple type inference for first-class polymorphism. In *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, James Hook and Peter Thiemann (Eds.). ACM, 283–294. https://doi.org/10.1145/1411204.1411245

Andres Löh, Conor McBride, and Wouter Swierstra. 2010. A Tutorial Implementation of a Dependently Typed Lambda Calculus. *Fundam. Informaticae* 102, 2 (2010), 177–207. https://doi.org/10.3233/FI-2010-304

Henry Mercer, Cameron Ramsay, and Neel Krishnaswami. 2022. Implicit Polarized F: local type inference for impredicativity. *CoRR* abs/2203.01835. https://doi.org/10.48550/ARXIV.2203.01835 arXiv:2203.01835

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Vol. 32. Chalmers University of Technology.

Martin Odersky, Christoph Zenger, and Matthias Zenger. 2001. Colored local type inference. In *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, Chris Hankin and Dave Schmidt (Eds.). ACM, 41–53. https://doi.org/10.1145/360204.360207

Benjamin C. Pierce and David N. Turner. 2000. Local type inference. *ACM Trans. Program. Lang. Syst.* 22, 1 (2000), 1–44. https://doi.org/10.1145/345099.345100

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, Chandra Krintz and Emery D. Berger (Eds.). ACM, 522–538. https://doi.org/10.1145/2908080.2908093

François Pottier and Yann Régis-Gianas. 2006. Stratified type inference for generalized algebraic data types. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, J. Gregory Morrisett and Simon L. Peyton Jones (Eds.). ACM, 232–244. https://doi.org/10.1145/1111037.1111058

Garrel Pottinger. 1980. A type assignment for the strongly normalizable λ-terms. *To HB Curry: essays on combinatory logic, lambda calculus and formalism* (1980), 561–577.

John C. Reynolds. 1991. The Coherence of Languages with Intersection Types. In *Theoretical Aspects of Computer Software, International Conference TACS '91, Sendai, Japan, September 24-27, 1991, Proceedings (Lecture Notes in Computer Science,*

Vol. 526), Takayasu Ito and Albert R. Meyer (Eds.). Springer, 675–700. https://doi.org/10.1007/3-540-54415-1_70

John C. Reynolds. 1997. Design of the Programming Language Forsythe. In *Algol-like Languages*, Peter W. O'Hearn and Robert D. Tennent (Eds.). Birkhäuser Boston, Boston, MA, 173–233. https://doi.org/10.1007/978-1-4612-4118-8_9

Nick Rioux, Xuejing Huang, Bruno C. d. S. Oliveira, and Steve Zdancewic. 2023. A Bowtie for a Beast: Overloading, Eta Expansion, and Extensible Data Types in F⋈. *Proc. ACM Program. Lang.* 7, POPL (2023), 515–543. https://doi.org/10.1145/3571211

Alejandro Serrano, Jurriaan Hage, Simon Peyton Jones, and Dimitrios Vytiniotis. 2020. A quick look at impredicativity. *Proc. ACM Program. Lang.* 4, ICFP (2020), 89:1–89:29. https://doi.org/10.1145/3408971

Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. 2018. Guarded impredicative polymorphism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 783–796. https://doi.org/10.1145/3192366.3192389

Hongwei Xi and Frank Pfenning. 1999. Dependent Types in Practical Programming. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, Andrew W. Appel and Alex Aiken (Eds.). ACM, 214–227. https://doi.org/10.1145/292540.292560

Ningning Xie and Bruno C. d. S. Oliveira. 2018. Let Arguments Go First. In *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 10801)*, Amal Ahmed (Ed.). Springer, 272–299. https://doi.org/10.1007/978-3-319-89884-1_10

Xu Xue and Bruno C. d. S. Oliveira. 2024. Contextual Typing (Artifact). Zenodo. https://doi.org/10.5281/zenodo.11429428

Xu Xue, Bruno C. d. S. Oliveira, and Ningning Xie. 2022. Applicative Intersection Types. In *Programming Languages and Systems - 20th Asian Symposium, APLAS 2022, Auckland, New Zealand, December 5, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13658)*, Ilya Sergey (Ed.). Springer, 155–174. https://doi.org/10.1007/978-3-031-21037-2_8

Jinxu Zhao, Bruno C. d. S. Oliveira, and Tom Schrijvers. 2019. A mechanical formalization of higher-ranked polymorphic type inference. *Proc. ACM Program. Lang.* 3, ICFP (2019), 112:1–112:29. https://doi.org/10.1145/3341716

Received 2024-02-28; accepted 2024-06-18