A Co-contextual Formulation of Type Rules and Its Application to Incremental Type Checking

Sebastian Erdweg[1] Oliver Bračevac[1] Edlira Kuci[1] Matthias Krebs[1] Mira Mezini[1,2]

[1]TU Darmstadt, Germany [2]Lancaster University, UK

Abstract

Type rules associate types to expressions given a typing context. As the type checker traverses the expression tree top-down, it extends the typing context with additional context information that becomes available. This way, the typing context coordinates type checking in otherwise independent subexpressions, which inhibits parallelization and incrementalization of type checking.

We propose a co-contextual formulation of type rules that only take an expression as input and produce a type and a set of *context requirements*. Co-contextual type checkers traverse an expression tree bottom-up and merge context requirements of independently checked subexpressions. We describe a method for systematically constructing a co-contextual formulation of type rules from a regular context-based formulation and we show how co-contextual type rules give rise to incremental type checking. Using our method, we derive incremental type checkers for PCF and for extensions that introduce records, parametric polymorphism, and subtyping. Our performance evaluation shows that co-contextual type checking has performance comparable to standard context-based type checking, and incrementalization can improve performance significantly.

Categories and Subject Descriptors F.3.1 [*Specifying and Verifying and Reasoning about Programs*]; F.3.2 [*Semantics of Programming Languages*]: Program analysis; F.4.1 [*Mathematical Logic*]: Lambda calculus and related systems

Keywords type checking; type inference; incremental; co-contextual; constraints; tree folding

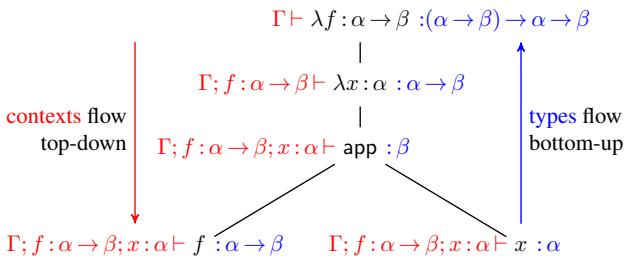
1. Introduction

Type checkers of modern programming languages play an important role in assuring software quality as they try to statically prove increasingly strong invariants over programs.

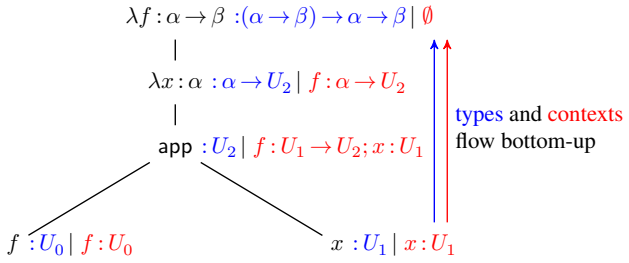
Typically, a type checker starts processing at the root node of a syntax tree and takes a typing context as an additional input to coordinate between sub-derivations by assigning types to jointly used variables. While traversing down the tree, the type checker extends the context with type bindings, making them available in subexpressions. When the type checker reaches a variable, it looks up the corresponding binding in the typing context. Since variables constitute leaves of the syntax tree, the downward traversal ends. The type checker then propagates the derived types back upward the syntax tree. In summary, while types flow bottom-up, typing contexts flow top-down – overall, the type system runs in a "down-up" mode.

The top-down context propagation of contexts has some problems. It hampers parallel and incremental type checking as well as compositional type checking because the typing of subexpressions depends on the typing of parent expressions (to retrieve the context) and vice versa (to retrieve the types). In practice, type checking may take considerable time. Especially for large software systems, it is important to reduce the run time of type checking to enable short feedback cycles [14]. To this end, some forms of incremental type checking find increasing interest in industry and, for example, are used in Eclipse's JDT and Facebook's Flow and Hack. In this paper, we present a generic method for constructing incremental type systems from standard type systems.

To enable type checkers with fine-grained incrementalization, we propose to eliminate top-down propagated contexts and to replace them by the dual concept of bottom-up propagated *context requirements*. We call such a type checking *co-contextual*. A co-contextual type checker starts processing at the leaves. Instead of looking up variables in a context, it invents fresh type variables as placeholders for their actual types. In addition, the co-contextual type checker generates context requirements which demand that the variables are actually bound with this type. As the co-contextual type checker



(a) Contextual type checking propagates contexts top-down.



(b) Co-contextual type checking propagates contexts bottom-up.

Figure 1. Contextual and co-contextual type checking.

progresses up the syntax tree, it refines the types of subexpressions and merges their context requirements.

To give an intuition about our proposal, we consider the process of type checking the simply-typed expression $\lambda f : \alpha \to \beta.\ \lambda x : \alpha.\ f\ x$ (α and β are arbitrary but fixed types) with an ordinary and a co-contextual type checker. Figure 1(a) depicts the ordinary type checking. It shows the syntax tree of the expression explicitly, marking the application term with app. We attach typing contexts and types to syntax tree nodes on their left-hand and right-hand side, respectively. The type attached to a node represents the type of the whole subexpression.

Figure 1(b) depicts the process of type checking the same expression by a co-contextual type checker. We separate a node's type T and context requirements R by a vertical bar $T \mid R$. The following tree illustrates how a co-contextual type check enables type checking a program bottom-up.

For the application term, the type checker refines U_0 to be a function type from U_1 to a fresh type variable U_2, which becomes the result type of the application term. Next, the type checker collects all context requirements of subexpressions, applies the type substitution $\{U_0 \mapsto U_1 \to U_2\}$ to them, and propagates them upward. This changes the type of f in the requirements. A λ-abstraction eliminates context requirements on the bound variable. When reaching the λ-abstraction of x, the type checker matches the required type for x with the declared type of x and propagates only the remaining requirement on f upward. Finally, the co-contextual type checker finds the same type as the context-based type checker above, and all context requirements have been satisfied. Any remaining context requirements in the root node of a program indicate a type error.

To see how co-contextual type checking avoids contextual coordination, consider the program $\lambda x : U_0.\ x\ x$. A context-based type checker propagates a single context $\Gamma; x : U_0$ to all subexpressions of the λ-abstraction, thus making sure that U_0 is used consistently as the type of x. In contrast, a co-contextual type checker generates context requirements $x : U_1$ and $x : U_2$ for the two variable nodes independently. Only when merging these requirements in the application term $x\ x$, the type checker coordinates the type of x by refining $U_1 = U_2$. In our example, this leads to a type error because type checking the application term also refines U_1 to a function type $U_2 \to U_3$. Thus, the type checker derives the unsolvable constraint $U_2 = U_2 \to U_3$ and correctly indicates a type error.

Since co-contextual type checkers do not coordinate subderivations, it is possible to incrementalize them systematically by applying memoization, incremental constraint solving, and eager substitution. Such incrementalization is independent of the module structure of the program.

In summary, we make the following contributions:

- We describe a generic method for constructing co-contextual type rules from traditional context-based type rules.
- We derive co-contextual type systems for PCF and for extensions that handle records, parametric polymorphism, and subtyping.
- We describe how co-contextual type systems give rise to incremental type checking, given that the type rules are in algorithmic form.
- We explain how to implement incremental type checking efficiently and describe implementations for PCF and its extensions.
- We evaluate the non-incremental and incremental performance of the co-contextual PCF type checker by comparison to a context-based algorithm.

2. Constructing Co-Contextual Type Systems

We can systematically construct co-contextual type rules from context-based type rules. The core idea is to eliminate the context and its propagation and instead to introduce context requirements that are propagated upward the typing derivation. The expressions and types involved in a type rule do not change. In this section, we illustrate how to derive a co-contextual type system for PCF. We use the following syntax for the expressions, types, and contexts of PCF:

$$\text{T-Num}\ \frac{}{\Gamma \vdash n : \mathit{Num} \mid \emptyset} \qquad \text{T-Add}\ \frac{\Gamma \vdash e_1 : T_1 \mid C_1 \quad \Gamma \vdash e_2 : T_2 \mid C_2}{\Gamma \vdash e_1 + e_2 : \mathit{Num} \mid C_1 \cup C_2 \cup \{T_1 = \mathit{Num}, T_2 = \mathit{Num}\}}$$

$$\text{T-Var}\ \frac{\Gamma(x) = T}{\Gamma \vdash x : T \mid \emptyset} \qquad \text{T-Abs}\ \frac{\Gamma; x : T_1 \vdash e : T_2 \mid C}{\Gamma \vdash \lambda x : T_1.\ e : T_1 \to T_2 \mid C} \qquad \text{T-App}\ \frac{\Gamma \vdash e_1 : T_1 \mid C_1 \quad \Gamma \vdash e_2 : T_2 \mid C_2 \quad U \textit{ is fresh}}{\Gamma \vdash e_1\ e_2 : U \mid C_1 \cup C_2 \cup \{T_1 = T_2 \to U\}}$$

$$\text{T-Fix}\ \frac{\Gamma \vdash e : T \mid C \quad U \textit{ is fresh}}{\Gamma \vdash \mathit{fix}\ e : U \mid C \cup \{T = U \to U\}} \qquad \text{T-If0}\ \frac{\Gamma \vdash e_1 : T_1 \mid C_1 \quad \Gamma \vdash e_2 : T_2 \mid C_2 \quad \Gamma \vdash e_3 : T_3 \mid C_3}{\Gamma \vdash \mathit{if0}\ e_1\ e_2\ e_3 : T_2 \mid C_1 \cup C_2 \cup C_3 \cup \{T_1 = \mathit{Num}, T_2 = T_3\}}$$

Figure 2. A contextual constraint-based formulation of the type system of PCF.

Contextual	Co-contextual
Judgment $\Gamma \vdash e : T \mid C$	Judgment $e : T \mid C \mid R$
Context syntax $\Gamma ::= \emptyset \mid \Gamma; x : T$	Requirements $R \subset x \times T$ map variables to their types
Context lookup $\Gamma(x) = T$	Requirement introduction $R = \{x : U\}$ with fresh unification variable U
Context extension $\Gamma; x : T$	Requirement satisfaction $R - x$ if $(R(x) = T)$ holds
Context duplication $\Gamma \to (\Gamma, \Gamma)$	Requirement merging $\mathit{merge}(R_1, R_2) = R\|_C$ if all constraints $(T_1 = T_2) \in C$ hold
Context is empty $\Gamma = \emptyset$	No unsatisfied requirements $R \stackrel{!}{=} \emptyset$

Figure 3. Operations on contexts and their co-contextual correspondence.

$$\begin{array}{lll} e & ::= & n \mid x \mid \lambda x : T.\ e \mid \mathit{fix}\ e \\ & & \mid e\ e \mid e + e \mid \mathit{if0}\ e\ e\ e \\ T & ::= & \mathit{Num} \mid T \to T \\ \Gamma & ::= & \emptyset \mid \Gamma; x : T \end{array} \qquad \begin{array}{l} \text{expressions} \\ \\ \text{types} \\ \text{typing contexts} \end{array}$$

Let us consider the type rule for variables first. Traditionally, the PCF rule for variables looks like this:

$$\text{T-Var}\ \frac{\Gamma(x) = T}{\Gamma \vdash x : T}$$

To co-contextualize this rule, we have to define it without a context. As a consequence, we cannot determine the type T of variable x, which depends on the binding and the usage context of x. To resolve this situation and to define a co-contextual type rule for variables, we apply a trick known from type inference: We generate a fresh type variable and use it as placeholder for the actual type of x. When more information about the type of x becomes available later on during type checking, we use type constraints and unification to retroactively refine the type of variable references. For instance, in the example from Section 1, we use fresh type variables U_0 and U_1 for the variable references f and x. Later, when checking the application of f to x, we refine the type of f by adding the type constraint $U_0 = U_1 \to U_2$, where U_2 is a fresh type variable itself. As this shows, a co-contextual type checker discovers type refinements in the form of constraints during type checking. Throughout the paper, we use the metavariable U for type variables that are unification variables and we use the metavariable X for user-defined type variables when they occur later on:

$$T ::= \ldots \mid U \qquad \text{unification type variables}$$

Reformulating type rules such that they produce type constraints instead of performing the actual type check is a standard technique in the context of type inference and type reconstruction [19]. In this work, we assume that the original type rules are given in a constraint-based style and take this as starting point for deriving co-contextual type rules. That is, we assume the original typing judgment has the form $\Gamma \vdash e : T \mid C$, where Γ is the typing context, e is the expression under analysis, and T is the type of e if all type constraints in set C hold. For reference, Figure 2 shows the constraint-based contextual type rules of PCF, where type constraints take the form of equalities:

$$c \in C ::= T = T \qquad \text{type constraints}$$

Co-contextual type rules use judgments of the form $e : T \mid C \mid R$, where e is the expression under analysis and T is the type of e if all type constraints in set C hold and all requirements in set R are satisfied. We can systematically construct co-contextual type rules for PCF from constraint-based contextual type rules through dualization. That is, we replace downward-propagated contexts with upward-propagated context requirements and we replace operations on contexts by their dual operations on context requirements as described in the following subsection.

2.1 Co-Contextual Syntax and Operations

We propose to use duality as a generic method for deriving co-contextual type systems. Figure 3 summarizes contextual and co-contextual syntax and operations for PCF. The syntax of context requirements is analogous to the syntax of typing

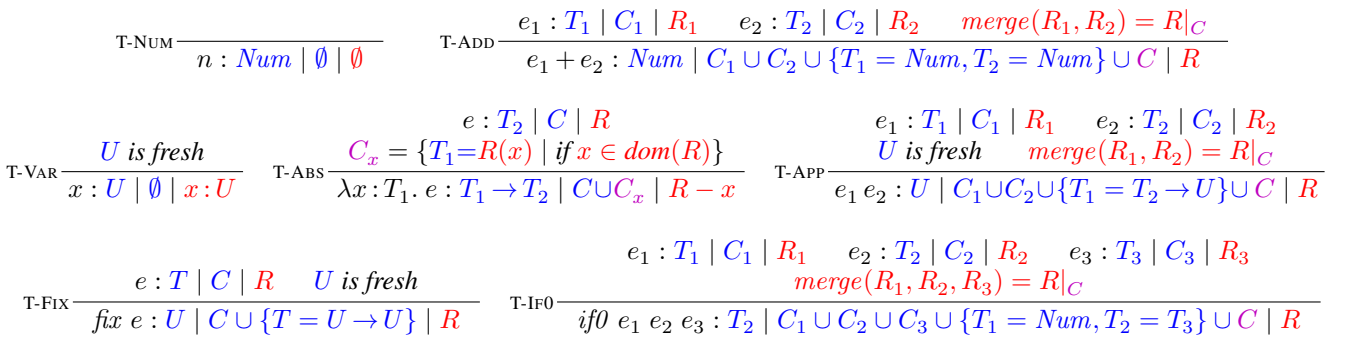


Figure 4. A co-contextual constraint-based formulation of the type system of PCF.

contexts. We represent context requirements as a set of type bindings $x : T$. Importantly, context requirements are not ordered and we maintain the invariant that there is at most one binding for x in any context-requirements set.

The first contextual operation is context lookup, which we translate into the dual operation of introducing a new context requirement. The context requirement declares that variable x must be well-typed and has type U, which is a fresh unification variable. Note the difference between co-contextual type checking and traditional type inference: Type inference generates a single fresh unification variable U when variable x is introduced (for example, by a λ-abstraction) and coordinates the typing of x via the context. In contrast, a co-contextual type system generates a new fresh unification variable for every reference of x in the syntax tree. Consequently, co-contextual type checkers typically produce more unification variables than context-based type inference, but they require no coordination.

The second operation is the extension of a context Γ with a new binding $x : T$. The co-contextual operation must perform the dual operation on context requirements, that is, eliminate a context requirement and reduce the set of context requirements. When eliminating a context requirement, it is important to validate that the requirement actually is satisfied. To this end, a co-contextual type system must check that the type of x that is required by the context requirements $R(x)$ is equivalent to T, the type that the original type rule assigned to x. If the constraint solver later finds that $R(x) = T$ does not hold, the type system has identified a context requirement that does not match the actual context. This indicates a type error.

The third operation is the duplication of a typing context, typically to provide it as input to multiple premises of a type rule. Context duplication effectively coordinates typing in the premises. The dual, co-contextual operation merges the context requirements of the premises, thus computing a single set of context requirements to propagate upward. Since the context requirements of the premises are generated independently, they may disagree on requirements for variables that occur in multiple subderivations. Accordingly, it is necessary to retroactively assure that the variables get assigned the same type. To this end, we use an auxiliary function $merge(R_1, R_2)$ that identifies overlapping requirements in R_1 and R_2 and generates a merged set of requirements R and a set of type-equality constraints C:

$$
\begin{array}{l}
merge(R_1, R_2) = R|_C \\
\textit{where} \quad R = R_1 \cup \{x : R_2(x) \mid x \in dom(R_2) \setminus dom(R_1)\} \\
\qquad\quad\ \ C = \{R_1(x) = R_2(x) \mid x \in dom(R_1) \cap dom(R_2)\}
\end{array}
$$

Function $merge$ is defined such that the merged requirements R favor R_1 in case of an overlap. This choice is not essential since it gets an equality constraint $R_1(x) = R_2(x)$ for each overlapping x anyways. Based on $merge$, we assume the existence of an n-ary function also called $merge$ that takes n requirement sets as input and merges all of them, yielding a single requirements set and a set of constraints. For more advanced type systems, we will need to refine function $merge$ (see Section 3).

The final operation to consider is the selection of an empty context. An empty context means that no variables are bound. For example, this occurs when type checking starts on the root expression. Dually, we stipulate that no context requirements may be left unsatisfied. Note that while the contextual operation selects the empty context, the co-contextual counterpart asserts that subderivations yield an empty requirement set. The difference is that a contextual type check fails when an unbound reference is encountered, whereas the co-contextual type check fails when the context requirement of an unbound variable cannot be satisfied.

We defined the translation from contextual operations to co-contextual operations in a compositional manner by applying duality. As a result, our translation is applicable to compound contextual operations, such as duplicating an extended context or extending an empty context to describe a non-empty initial context.

2.2 Co-Contextual PCF

Figure 4 shows the co-contextual type rules of PCF, which we derived according to the method described above. The derivation of T-Num and T-Fix is straightforward as neither rule involves any context operation. In rule T-Var, dual to context lookup, we require x is bound to type U in the context. In rule T-Abs, dual to context extension with x, we check if variable x has been required by the function body e. If there is a requirement for x, we add the constraint $T_1{=}R(x)$ and remove that requirement $R - x$. Otherwise C_x is empty and we only propagate the constraints of the body.

The remaining rules T-Add, T-App, and T-If0 use requirement merging dual to the duplication of contexts in the original rules. Each merge gives rise to an additional set of constraints that we propagate. Note that in T-If0 we have to merge requirements from all three subexpressions.

We can use the co-contextual type rules to compute the type of PCF expressions. Given an expression e and a derivation $e : T_e \mid C \mid R$, e is well-typed if R is empty and the constraint system C is solvable. If e is well-typed, let $\sigma : U \to T$ be a type substitution that solves the constraint system C. Then, the type of e is $\sigma(T_e)$.

In Section 1, we showed an example derivation of co-contextual type checking the expression $\lambda f : \alpha \to \beta.\ (\lambda x : \alpha.\ f\ x)$. For presentation's sake, we eagerly solved constraints and applied the resulting substitutions. Actually, the co-contextual PCF type system defined here generates the constraint set $\{U_0 = U_1 \to U_2, U_1 = \alpha, U_0 = \alpha \to \beta\}$ and derives the result type $T = (\alpha \to \beta) \to \alpha \to U_2$. Subsequent constraint solving yields the substitution $\sigma = \{U_0 \mapsto (\alpha \to \beta), U_1 \mapsto \alpha, U_2 \mapsto \beta\}$ and the final result type $\sigma(T) = (\alpha \to \beta) \to \alpha \to \beta$.

We prove that our co-contextual formulation of PCF is equivalent to PCF. To relate derivations on open expressions containing free variables, we demand the context requirements to be a subset of the provided typing context. In particular, we get equivalence for closed expressions by setting $\Gamma = \emptyset$ and $R = \emptyset$. In our formulation, we call a syntactic entity ground if it does not contain unification variables and we write $\Gamma \supseteq R$ if $\Gamma(x) = R(x)$ for all $x \in \mathit{dom}(R)$. The following theorem establishes the equivalence of the two formulations of PCF:

Theorem 1. *A program e is typeable in contextual PCF if and only if it is typeable in co-contextual PCF:*

$\Gamma \vdash e : T \mid C$ *and* $\mathit{solve}(C) = \sigma$ *such that* $\sigma(T)$ *and* $\sigma(\Gamma)$ *are ground*

if and only if

$e : T' \mid C' \mid R$ *and* $\mathit{solve}(C') = \sigma'$ *such that* $\sigma'(T')$ *and* $\sigma'(R)$ *are ground*

If e is typeable in contextual and co-contextual PCF as above, then $\sigma(T) = \sigma'(T')$ and $\sigma(\Gamma) \supseteq \sigma'(R)$.

Proof. By structural induction on e. A detailed proof appears in Appendix A. □

$$\text{T-Record}\ \frac{e_i : T_i \mid C_i \mid R_i \ \text{ for } i \in 1 \ldots n \qquad \mathit{merge}(R_1, \ldots, R_n) = R|_{C_R} \quad \bigcup_{i \in 1 \ldots n} C_i = C}{\{l_i = e_i\}_{i \in 1 \ldots n} : \{l_i : T_i\}_{i \in 1 \ldots n} \mid C \cup C_R \mid R}$$

$$\text{T-Lookup}\ \frac{e : T \mid C \mid R \qquad U \ \textit{is fresh}}{e.l : U \mid C \cup \{T.l = U\} \mid R}$$

Figure 5. Co-contextual type rules for records.

3. Extensions of PCF

In this section, we present co-contextual type systems for extensions of PCF with records, parametric polymorphism, and subtyping.

3.1 Simple Extensions: Records

Many type-system features do not change or inspect the typing context. We can define such features as simple extensions of the co-contextual PCF type system. As a representative for such features, we present an extension with records, using the following extended syntax for expressions, types, and type constraints:

e	$::=$	$\ldots \mid \{l_i = e_i\}_{i \in 1 \ldots n} \mid e.l$	record expressions
T	$::=$	$\ldots \mid \{l_i : T_i\}_{i \in 1 \ldots n}$	record types
c	$::=$	$\ldots \mid T.l = T$	field type constraint

The additional type rules for records appear in Figure 5. Type rule T-Record defines how to type check record expressions $\{l_i = e_i\}_{i \in 1 \ldots n}$. The type of the record expression is a record type, where each label is associated type T_i of subexpression e_i. To type check the subexpressions e_i, a traditional contextual type rule for record expressions uses a replica of its typing context for each subderivation. In accordance with the definitions in Figure 3, a co-contextual type rule for record expressions merges the requirements of all subderivations. Type rule T-Record in Figure 5 merges the requirements R_i into the fully merged requirements set R with additional constraints C_R. We propagate the merged requirements, the additional constraints, and the original constraints C_i of the subderivations.

Type rule T-Lookup defines type checking for field lookup $e.l$. In our setting, we cannot simply match the type T of e to extract the type of field l because T may be a unification variable that is only resolved later after solving the generated constraints. Instead, we use a constraint $T.l = U$ that expresses that T is a record type that contains a field l of type U.

Similar to records, we can easily define further simple extensions of our co-contextual formulation of PCF, such as variants or references.

$$\text{T-TABS}\ \frac{e : T \mid C \mid (R, R^T)}{\lambda X.\, e : \forall X.T \mid C \mid (R, R^T - X)} \qquad \text{T-TAPP}\ \frac{e : T_1 \mid C \mid (R, R_e^T) \quad U, U_b, U_r \textit{ is fresh} \quad \vdash T\ \mathsf{ok} \mid R^T}{e[T] : U_r \mid C \cup \{T_1 = \forall U.\, U_b\} \cup \{U_r = \{U \mapsto T\} U_b\} \mid (R, R_e^T \cup R^T)}$$

$$\text{T-ABS}\ \frac{e : T_2 \mid C \mid (R, R_e^T) \quad \vdash T_1\ \mathsf{ok} \mid R_1^T \quad C_x = \{T_1 = R(x) \mid \textit{if } x \in \textit{dom}(R)\}}{\lambda x : T_1.\, e : T_1 \to T_2 \mid C \cup C_x \mid (R - x, R_e^T \cup R_1^T)} \qquad \text{T-APP}\ \frac{e_1 : T_1 \mid C_1 \mid (R_1, R_1^T) \quad e_2 : T_2 \mid C_2 \mid (R_2, R_2^T) \quad U \textit{ is fresh} \quad \textit{merge}(R_1, R_2) = R|_C}{e_1\, e_2 : U \mid C_1 \cup C_2 \cup \{T_1 = T_2 \to U\} \cup C \mid (R, R_1^T \cup R_2^T)}$$

Figure 6. Co-contextual type rules for parametric polymorphism.

3.2 Parametric Polymorphism

In the following, we present the co-contextual formulation of PCF extended with parametric polymorphism. This extension is interesting with respect to our co-contextual formulation because (i) the type checker can encounter type applications without knowledge of the underlying universal type and (ii) parametric polymorphism requires further context operations to ensure that there are no unbound type variables in a program. To support parametric polymorphism, we first add new syntactic forms for type abstraction and application as well as for type variables and universal types.

$$\begin{array}{rcl} e & ::= & \ldots \mid \lambda X.\, e \mid e[T] \\ T & ::= & \ldots \mid X \mid \forall X.\, T \mid \forall U.\, T \\ c & ::= & \ldots \mid T = \{X \mapsto T\} T \mid T = \{U \mapsto T\} T \end{array}$$

Note that due to the constraint-based nature of co-contextual type checking, we require support for universal types that quantify over user-supplied type variables X as well as unification type variables U. Importantly, the universal type $\forall U.\, T$ does not bind the unification variable U; unification variables are always bound globally. Instead, $\forall U.\, T$ binds the type variable that will eventually replace U. Moreover, we require new constraints of the form $T_1 = \{X \mapsto T_2\} T_3$ that express that T_1 is the result of substituting T_2 for X in T_3. We define similar constraints for substituting unification variables. However, their semantics differ in that the constraint solver must delay the substitution of a unification variable until it is resolved to a proper type. For example, the substitution in $T_1 = \{U \mapsto T_2\} X$ must be delayed because it might later turn out that $U = X$. Furthermore, the constraint solver may not substitute user-supplied type variables X as they are not unification variables, hence the constraint $X_1 = X_2 \to X_2$ does not hold. This distinction of type variables also entails specific rules for the substitution and unification of universal types, which permits the refinement of unification variables even when they appear as binding or bound occurrences.

Since parametric polymorphism introduces bindings for type variables, we also need to track that no unbound type variables occur in a program. A traditional contextual type checker adds bound type variables to the typing context and checks that all occurrences of type variables are indeed bound. We can use the same strategy as for term variables (Section 2) to co-contextualize type-variable handling. In particular, we introduce an additional requirements component for type variables $R^T \subset X$ and extend our typing judgment $e : T \mid C \mid (R, R^T)$ to propagate required type variables R^T. Dual to lookup and introduction of type variables in contextual type checking, we produce type-variable requirements when checking a user-supplied type for well-formedness $\vdash T\ \mathsf{ok} \mid R^T$ and we eliminate type-variable requirements when binding a type variable in a type-level λ-abstraction. As before, an expression is only well-typed if all requirements are satisfied, that is, there are neither term-variable nor type-variable requirements on the root of the syntax tree.

Figure 6 shows the type rules for type abstraction, type application, and term abstraction. Rule T-TABS handles type abstraction $\lambda X.\, e$. It eliminates type-variable requirements on the bound type variable X and propagates the remaining type-variable requirements $R^T - X$. Rule T-TAPP handles type applications $e[T]$. It checks the subexpression e for well-typedness and the application type T for well-formedness and propagates their combined type-variable requirements $R_e^T \cup R^T$. As the first constraint of T-TAPP stipulates, type application is only well-typed if the type of e is a universal type $\forall U.\, U_b$. The type of the type application then is the result of substituting T for U in U_b, as the second constraint defines.

Type rule T-ABS in Figure 6 is an extended version of the co-contextual PCF rule for λ-abstraction from Figure 4. Due to the existence of type variables, we added a premise that checks the well-formedness of the type annotation T_1. We propagate the resulting type-variable requirements together with the type-variable requirements of the function body. Finally, type rule T-APP illustrates how to extend all other rules of PCF such that they merge and propagate type-variable requirements from subexpressions. Note that due to the simplicity of type-variable requirements, the merge operation is simply set union. We would require a more sophisticated merge operation when introducing type variables of different kinds, for example, to realize higher-order polymorphism.

To illustrate these type rules, consider the co-contextual type checking of the polymorphic identity function instantiated for Num: $(\lambda X.\, \lambda x : X.\, x)[Num]$.

$$\mathsf{tapp}\ [Num] : U_r \mid \left\{ \begin{array}{l} U_0 = X, \\ \forall X.\,(X \to U_0) = \forall U.\,U_b, \\ U_r = \{U \mapsto Num\} U_b \end{array} \right\} \mid (\emptyset, \emptyset)$$

$$\lambda X : \forall X.\,(X \to U_0) \mid \{U_0 = X\} \mid (\emptyset, \emptyset)$$

$$\lambda x : X : X \to U_0 \mid \{U_0 = X\} \mid (\emptyset, \{X\})$$

$$x : U_0 \mid \emptyset \mid (x : U_0, \emptyset)$$

The type checker processes the expression bottom-up. First, it associates a fresh unification variable U_0 to x. Second, the λ-abstraction binds x to type X, which yields the constraint $U_0 = X$ and a type-variable requirement on X. Third, this requirement is immediately discharged by the type-level λ-abstraction that binds X. Finally, the type application rule requires a universal type and computes the result type U_r via a type-substitution constraint. Subsequent constraint solving yields the substitution $\sigma = \{U_0 \mapsto X, U \mapsto X, U_b \mapsto (X \to X), U_r \mapsto (Num \to Num)\}$.

Our type system rejects expressions with unbound type variables. For example, the expression $\lambda f : Num \to X.\ x\,0$ contains an unbound type variable X. When type checking this expression in our type system, we receive an unsatisfied type-variable requirement that represents this error precisely. Furthermore, despite using constraints, our type system correctly prevents any refinement of universally quantified type variables. For example, our type system correctly rejects the expression $\lambda X.\ \lambda x : X.\ x + x$, which tries to refine X to Num to perform addition.

Type inference and let polymorphism. As discussed in the previous section, co-contextual type checking is different from type inference in that co-contextual type checking avoids typing contexts and generates a fresh unification variable for each variable reference in the syntax tree. Nevertheless, co-contextual type checking can infer types just as well, because it already computes principal types.

For example, to support type inference for a let-polymorphic type system, four changes to our rules from Figure 6 are necessary. First, remove rules T-TAbs and T-TApp and the corresponding expression syntax, because type abstraction and application is inferred. Second, remove the type annotation in T-Abs and use the type $R(x)$ as argument type instead. If $R(x)$ is undefined, use a fresh unification variable. Third, add a *let* construct and type rule T-Let that collects all free type variables in the type of the bound expression and constructs a typing schema. Due to constraint-based typing, the schema construction needs to be deferred until the type of the bound expression does not contain any unification variables (otherwise one might miss some free type variables). Define a new kind of constraint to express precisely that. Fourth, change T-App to use a new kind of constraint for instantiating type schemas, instead of checking for equality of the argument type directly.

3.3 Subtyping

As a final extension, we consider a co-contextual formulation of PCF with subtyping. Subtyping is an interesting extension because it affects the semantics of a typing contexts and, hence, context requirements. In particular, subtyping weakens the assumptions about variable bindings $x : T$ in typing contexts. In standard PCF, $(x : T_x; \Gamma)$ means that variable x has *exactly* type T_x. In contrast, in PCF with subtyping, $(x : T_x; \Gamma)$ means that T_x is *an upper bound* of the type of values substitutable for x: All values must at least adhere to T_x. Dually, a type annotation T_x on a λ-abstraction is a lower bound for the type required for x in subexpressions: Subexpressions can at most require T_x. Thus, subtyping affects the merging and satisfaction of context requirements.

We adapt the definition of $merge$ to correctly combine requirements from different subexpressions. Due to subtyping, different subexpressions can require different types on the same variable. In consequence, a variable has to simultaneously satisfy the requirements of all subexpressions that refer to it. That is, when we merge overlapping context requirements, we require the type of shared variables to be a subtype of both originally required types:

$$\begin{array}{lll} merge(R_1, R_2) = R|_C & & \\ \textit{where } X & = dom(R_1) \cap dom(R_2) & \\ & U_x = \textit{fresh variable for each } x \in X & \\ & R & = (R_1 - dom(R_2)) \cup (R_2 - dom(R_1)) \\ & & \quad \cup \{x : U_x \mid x \in X\} \\ & C & = \{U_x <: R_1(x), U_x <: R_2(x) \mid x \in X\} \end{array}$$

We do not stipulate a specific subtyping relation. However, we add new forms of type constraints with standard semantics to express subtyping, joins, and meets:

$$\begin{array}{lll} c ::= \ \ldots & \mid T <: T & \text{subtype constraint} \\ & \mid T = T \vee T & \text{least upper bound (join)} \\ & \mid T = T \wedge T & \text{greatest lower bound (meet)} \end{array}$$

A least upper bound constraint $T_1 = T_2 \vee T_3$ states that type T_1 is the least type in the subtype relation such that both T_2 and T_3 are subtypes of T_1. A greatest lower bound constraint $T_1 = T_2 \wedge T_3$ states that the type T_1 is the greatest type in the subtype relation such that both T_2 and T_3 are supertypes of T_1.

Figure 7 shows the co-contextual rules for PCF enriched with subtyping. Only the rules for λ-abstractions, applications, conditionals, and fixpoints change with respect to the co-contextual PCF type system in Figure 4. First, consider the rule for λ-abstraction T-Abs. As discussed above, a context requirement on x only describes an upper bound on the declared type of x. Or conversely, the declared type of a variable x is a lower bound on what subexpressions can require for x. Accordingly, we replace the type-equality constraint by a subtype constraint $T_1 <: R(x)$.

$$\text{T-Abs}\ \frac{e : T_2 \mid C \mid R \qquad C_x = \{T_1 <: R(x) \mid \textit{if } x \in \textit{dom}(R)\}}{\lambda x : T_1.\ e : T_1 \to T_2 \mid C \cup C_x \mid R - x}$$

$$\text{T-App}\ \frac{e_1 : T_1 \mid C_1 \mid R_1 \qquad e_2 : T_2 \mid C_2 \mid R_2 \qquad U_1, U_2 \textit{ is fresh} \qquad \textit{merge}(R_1, R_2) = R|_C}{e_1\ e_2 : U_2 \mid C_1 \cup C_2 \cup \{T_1 = U_1 \to U_2, T_2 <: U_1\} \cup C \mid R}$$

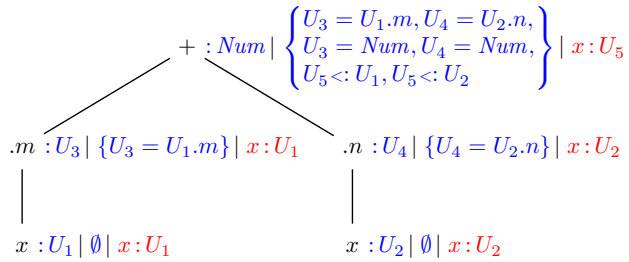
$$\text{T-If0}\ \frac{\begin{array}{c} e_1 : T_1 \mid C_1 \mid R_1 \qquad e_2 : T_2 \mid C_2 \mid R_2 \qquad e_3 : T_3 \mid C_3 \mid R_3 \\ \textit{merge}(R_1, R_2, R_3) = R|_{C_R} \qquad U \textit{ is fresh} \\ C = \{T_1 = \textit{Num}, U = T_2 \vee T_3\} \cup C_1 \cup C_2 \cup C_3 \cup C_R \end{array}}{\textit{if0}\ e_1\ e_2\ e_3 : U \mid C \mid R}$$

$$\text{T-Fix}\ \frac{e : T \mid C \mid R \qquad U_1, U_2 \textit{ is fresh}}{\textit{fix}\ e : U_1 \mid C \cup \{T = U_1 \to U_2, U_2 <: U_1\} \mid R}$$

Figure 7. Co-contextual type rules for PCF with subtyping.

The other rules T-App, T-If0, and T-Fix are straightforward extensions to allow for subtyping. In rule T-App, we allow the argument to be of a subtype of the function's parameter type as usual. Rule T-If0 declares the type of the conditional to be the least upper bound of the two branch types, which we express by the least upper bound constraint $U = T_1 \vee T_2$. In rule T-Fix, we permit the fixed function to produce values whose type is a subtype of the function's parameter type.

To illustrate co-contextual type checking with subtypes, consider PCF with records and the usual depth and width subtyping for records. When type checking the expression $e = x.m + x.n$ with free x, we get the following derivation:



We can simplify the resulting constraint set by eliminating U_3 and U_4 to get $\{\textit{Num} = U_1.m, \textit{Num} = U_2.n, U_5 <: U_1, U_5 <: U_2\}$, where U_5 is the type required for x. Importantly, the type of x must be a subtype of U_1 and U_2, which in turn must provide fields m and n respectively. Thus, the type U_5 of x must be a record type that at least provides fields m and n. Indeed, when we close above expression e as in $\lambda x : T.\ e$, type rule T-Abs yields another constraint $T <: U_5$. Accordingly, type checking succeeds for an annotation $T = \{m : \textit{Num}, n : \textit{Num}, o : \textit{Num} \to \textit{Num}\}$. But type checking correctly fails for an annotation $T = \{m : \textit{Num}\}$, because $T <: U_2$ by transitivity such that $\textit{Num} = U_2.n$ is not satisfiable.

4. Incremental Type Checking

Incremental computations often follow a simple strategy: Whenever an input value changes, transitively recompute all values that depend on a changed value until a fixed-point is reached [18]. We apply the same strategy to incrementalize type checking. As we will see, the avoidance of contextual coordination in co-contextual type systems makes dependency tracking particularly simple and thus enables incremental type checking.

For illustration, we use the following PCF expressions as a running example throughout this section. To simplify the example, we added subtraction to PCF in a straightforward manner.

$$\begin{aligned} \textit{mul} &= \textit{fix}\ (\lambda f : \textit{Num} \to \textit{Num} \to \textit{Num}. \\ &\qquad \lambda m : \textit{Num}.\ \lambda n : \textit{Num}.\ \textit{if0}\ m\ 0\ (n + f\ (m-1)\ n)) \\ \textit{notfac} &= \textit{fix}\ (\lambda f : \textit{Num} \to \textit{Num}. \\ &\qquad \lambda n : \textit{Num}.\ \textit{if0}\ (n-1)\ 1\ (\textit{mul}\ n\ (f\ (n-2)))) \end{aligned}$$

The first expression $\textit{mul}$ defines multiplication on top of addition by recursion on the first argument. The second expression $\textit{notfac}$ looks similar to the factorial function, but it is undefined for $n = 0$ and the recursive call subtracts 2 instead of 1 from n. Below, we exemplify how to incrementally type check these expression when changing $\textit{notfac}$ to match the factorial function. The initial type check of $\textit{mul}$ and $\textit{notfac}$ generate 11 and 12 constraints, respectively.

4.1 Basic Incrementalization Scheme

In the previous sections, we defined co-contextual type rules using a typing judgment of the form $e : T \mid C \mid R$. In this section, we additionally assume that the type rules are given in an algorithmic form, that is, they are syntax-directed and their output is fully determined by their input. Therefore, another way to understand the type rules is as a function $\textit{check}_1 : e \to T \times C \times R$ that maps an expression to its type, typing constraints, and context requirements.

Note that $\textit{check}_1$ only depends on the expression and is independent of any typing context. This means that an expression e is assigned the same type, constraints, and requirements independent of the usage context (modulo renaming of unification variables). For example, $\textit{check}_1(e)$ yields the same result no matter if it occurs as a subderivation of $e + 5$ or $\lambda x : \textit{Num}.\ e$. The only difference is that some usage scenarios may fail to satisfy the constraints or context requirements generated for e. Accordingly, when an expression changes, it is sufficient to propagate these changes up the expression tree to the root node; siblings are never affected.

Based on this observation, we can define a simple incrementalization scheme for co-contextual type checking:

> E = set of non-memoized subexpressions in *root*
> E^C = transitive closure of E under parent relationship
> E_p^C = sequentialize E^C in syntax-tree post-order
> for each $e \in E_p^C$
> recompute and memoize the type of e
> using the memoized types of its subexpressions

That is, we visit all subexpressions that have changed and all subexpressions that (transitively) depend on changed subexpressions. We use a post-order traversal (i.e., bottom-up) to ensure we visit all changed subexpressions of an expression before recomputing the expression's type. Accordingly, when we type check an expression in E_p^C, the types (and constraints and requirements) of all subexpressions are already available through memoization. We present an efficient implementation of this scheme in Section 5.

To illustrate, let us consider the example expression *notfac* from above. First, we observe that a change to *notfac* never affects the typing of *mul*, which we can fully reuse. When we change the *if*-condition of *notfac* from $n-1$ to n, our incremental type checker recomputes types for the expressions $E_p^C = \langle n_{cond}, \mathit{if0}, \lambda n, \lambda f, \mathit{fix}\rangle$, starting at the changed condition. Importantly, we reuse the memoized types from the initial type check of the *else*-branch. When instead changing the subtraction in the recursive call of *notfac* from $n-2$ to $n-1$, the type checker recomputes types for the expressions $E_p^C = \langle 1, n-1, \mathsf{app}_f, \mathsf{app}_{mul2}, \mathit{if0}, \lambda n, \lambda f, \mathit{fix}\rangle$. While this change entails lots of recomputation, we still can reuse results from previous type checks for the application of *mul* to its first argument and for the *if*-condition. Finally, consider what happens when we introduce a type error by changing the binding λn to λx. A contextual type checker would have to reconstruct the derivation of the body because the context changed and n now is unbound. In contrast, our incremental type checker can reuse all constraints from the body of the abstraction and only has to regenerate the constraint for *fix*. The table below summarizes the number of regenerated and reused constraints. Our performance evaluation in Section 6 confirms that incrementalization of type checking can improve type-checking performance significantly.

Changes in *notfac*	Regenerated constraints	Reused constraints	Reused from *mul*
if-cond. $n-1 \mapsto n$	4	6	11
rec. call $n-2 \mapsto n-1$	9	3	11
binding $\lambda n \mapsto \lambda x$	1	11	11

4.2 Incremental Constraint Solving

In the previous subsection, we have developed an incremental type checker $\mathit{check}_1 : e \to T \times C \times R$. However, all that check_1 actually does is to generate constraints and to collect context requirements. That is, even when run incrementally, check_1 yields a potentially very large set of constraints that we have to solve in order to compute the type of e. We only incrementalized constraint generation but not constraint solving so far.

Designing an incremental constraint solver is difficult because unification and substitution complicate precise dependency tracking and updating of previously computed solutions when a constraint is removed. However, since type rules only add constraints but never skip any constraints from subderivations, we can solve the intermediate constraint systems and propagate and compose their solutions. To this end, we assume the existence of two functions:

$$\begin{array}{lcll} \mathit{solve} & : & C & \to \quad \sigma \times C^- \times C^? \\ \mathit{finalize} & : & \sigma \times C^- \times C^? & \to \quad \sigma \times C^- \end{array}$$

Function *solve* takes a set of constraints and produces a partial solution $\sigma : U \to T$ mapping unification variables to types, a set of unsatisfiable constraints C^- indicating type errors, and a set of possibly satisfiable constraints $C^?$ that may or may not hold in a larger context. The intermediate solutions are not necessarily ground substitutions since domain U is a subset of domain T. While intermediate solutions discharge equality constraints, they do not necessarily eliminate all unification variables. Function *finalize* assumes a closed world and either solves the constraints in $C^?$ or marks them as unsatisfiable. We include $C^?$ for type systems that require (partially) context-sensitive constraint solving. For example, for PCF with subtyping, $C^?$ contains constraints that encode lower and upper bounds on type variables. In practice, to achieve good performance, it is important to normalize the constraints in $C^?$ and represent them compactly.

Using *solve*, we can modify type rules such that they immediately solve the constraints they generate. Type rules then propagate the composed solutions from their own constraints and all subderivations. For example, we obtain the following definition of T-App, where we write $\sigma_1 \circ \sigma_2$ to denote substitution composition.[1]

$$\textsc{T-App}\ \frac{\begin{array}{c} e_1 : T_1 \mid \sigma_1 \mid C_1^- \mid C_1^? \mid R_1 \\ e_2 : T_2 \mid \sigma_2 \mid C_2^- \mid C_2^? \mid R_2 \\ U \textit{ is fresh} \qquad \mathit{merge}(R_1, R_2) = R|_C \\ \mathit{solve}(C \cup \{T_1 = T_2 \to U\}) = (\sigma_3, C_3^-, C_3^?) \\ \sigma_1 \circ \sigma_2 \circ \sigma_3 = \sigma \end{array}}{e_1\, e_2 : U \mid \sigma \mid C_1^- \cup C_2^- \cup C_3^- \mid C_1^? \cup C_2^? \cup C_3^? \mid R}$$

This gives rise to an incremental type checker $\mathit{check}_2 : e \to T \times \sigma \times C^- \times C^? \times R$ using the incrementalization scheme from above. In contrast to check_1, check_2 actually conducts incremental constraint solving since we incorporated con-

[1] Similar to requirements merging, the composition of substitutions can yield additional constraints when the domains of the substitutions overlap, which can be easily resolved by additional constraint solving. We omit this detail to keep the presentation concise.

straint solving into the type rules. The only constraints we have to solve non-incrementally are those in $C^?$, for which we use *finalize* on the root node.

Let us revisit the example expression *notfac* from above. When changing the *if*-condition or recursive call, we obtain the same number of regenerated constraints. However, instead of reusing previously generated constraints, $check_2$ reuses previously computed solutions. This means that after a change, we only have to solve the newly generated constraints. For example, after the initial type check, we never have to solve the constraints of *mul* again, because it does not change. As our performance evaluation in Section 6 shows, incremental constraint solving significantly improves incremental performance and, somewhat surprisingly, also the performance of the initial type check.

4.3 Eager Substitution

Co-contextual type rules satisfy an interesting property that enables eager substitution of constraint solutions. Namely, the constraints in independent subexpressions yield non-conflicting substitutions.

This statement holds for two reasons. First, every time a type rule requires a fresh unification variable U, this variable cannot be generated fresh by any other type rule. Thus, U cannot occur in constraints generated while type checking any independent subexpression. Hence, there can be at most one type assigned to U by a constraint. Second, we formulated co-contextual type rules in a way that strictly separates user-defined type variables X from unification variables U generated by type rules. While a unification variable is generated fresh once, a user-defined type variable X can occur multiple times in the syntax tree, for example within type annotations of different λ-abstractions. Thus, type checking independent subexpressions can yield constraints that jointly constrain X. When we solve such constraints independently, there is the danger of assigning different types to X, which would require coordination. However, since the substitutions computed by *solve* map unification variables to types and user-defined type variables are not considered unification variables, this situation cannot occur.

$check_2$ propagated substitutions up the tree. However, as substitutions are non-conflicting, we can eagerly apply the substitution within the type rule, thus avoiding its propagation altogether. For example, we can redefine type rule T-App as follows:

$$\text{T-App}\ \frac{\begin{array}{c} e_1 : T_1 \mid C_1^- \mid C_1^? \mid R_1 \quad e_2 : T_2 \mid C_2^- \mid C_2^? \mid R_2 \\ U \textit{ is fresh} \qquad \textit{merge}(R_1, R_2) = R|_C \\ \textit{solve}(C \cup \{T_1 = T_2 \to U\}) = (\sigma, C_3^-, C_3^?) \\ C^- = C_1^- \cup C_2^- \cup C_3^- \qquad C^? = C_1^? \cup C_2^? \cup C_3^? \end{array}}{e_1\, e_2 : \sigma(U) \mid \sigma(C^-) \mid \sigma(C^?) \mid \sigma(R)}$$

In this type rule, the substitution σ is not propagated. Instead, we directly apply it to all components of the type rule's result. By applying the substitution, we eliminate all unification variables $U \in \textit{dom}(\sigma)$ from the result. But then, as unification variables are not shared between independent trees, there is no need to propagate the substitution itself.

This design yields an incremental type-check function $check_3 : e \to T \times C^- \times C^? \times R$. Compared to the previous version, this type checker can improve both the initial and incremental performance because it avoids the propagation, composition, and memoization of substitutions. As our performance evaluation in Section 6 shows, we achieve best performance by using a hybrid approach that stores and propagates small substitutions but eagerly applies larger substitutions that bind ten or more variables.

5. Technical Realization

We have developed efficient incremental type checkers in Scala for PCF and for each of the extensions described in Section 3. The source code is available online at

`https://github.com/seba--/incremental`.

In-tree memoization. Implementing incremental type checking as described in the previous section requires memoization to map expressions to previously computed typing information. When type checking an expression, we first check if a type has been previously memoized, in which case we return this type without further computation. For better memory and time performance, we do not use a lookup table, but memoize previously computed types directly in the syntax tree. As a consequence of in-tree memoization, trees that are equivalent but represented as separate objects do not share their memoized types. To avoid this, our representation supports syntax representations with sharing based on acyclic object graphs. To support tree changes efficiently, our expressions store a mutable list of child expressions. We encode an incremental change to a syntax tree via an update to the parent's list of child expressions. If the new child has a memoized type, we keep it and invalidate the parent's type. If the child has no memoized type, dependency tracking will invalidate the parent's type automatically.

Efficient incremental type checking. The incrementalization scheme described in Section 4.1 (1) selects all non-memoized subexpressions, (2) closes them under parent relationship, (3) orders them in syntax-tree post-order, and (4) iterates over them to recompute and memoize types. To implement this scheme efficiently, we merge these operations into a single post-order tree traversal. During the traversal, we recompute and memoize the type of a subexpression if the type was not computed before or if the type of any direct subexpression was recomputed during the same traversal. This traversal has the same semantics as the scheme presented before, but avoids the materialization and iteration over intermediate collections of subexpressions.

Constraint systems. The implementation details of the constraint systems heavily depend on the supported language constructs. To simplify the definition of incremental type check-

ers, our framework provides an abstract constraint-system component that captures the commonalities of different constraint systems, but abstracts from the internal representation and processing of constraints.

For PCF, the constraint system is straightforward and simply solves equational constraints by unification. The extensions of PCF introduce record-field type constraints, universal-type substitution constraints and subtype constraints. Some of these constraints cannot be immediately solved when they are generated but only later when more information about the constrained type variables is available. For good performance, we represent such constraints compactly and normalize them eagerly whenever new constraints are added to the constraint system. In particular, for subtyping, our constraint solver relies on ideas from local type inference [20] and subtype-constraint simplification [21]. That is, we keep track of the lower and upper bounds on type variables and gradually refine them. Type variables have a polarity (covariant, contravariant or invariant), which determines whether to maximize or minimize the type of a variable. We transitively normalize newly generated subtyping constraints to subtyping constraints on type variables.

6. Performance Evaluation

We evaluate the performance of non-incremental and incremental co-contextual type checking through micro-benchmarking. Specifically, we compare the performance of the following five PCF type checkers on a number of synthesized input programs:[2]

- DU: Standard contextual constraint-based down-up type checker (base line) that propagates contexts downward and types upward.
- BU1: Co-contextual bottom-up type checker with incremental constraint generation (Section 4.1).
- BU2: Like BU1 but with incremental constraint solving (Section 4.2).
- BU3: Like BU2 but with eager substitution (Section 4.3).
- BU4: Like BU3 but only eagerly substitute when $|\sigma| \geq 10$.

We expect to show two results with our evaluation. First, when running non-incrementally, our co-contextual type checkers have performance comparable to the standard contextual type checker DU. Second, after an incremental program change, our co-contextual type checkers outperform the standard contextual type checker DU. Moreover, our evaluation provides some initial data for comparing the performance of the co-contextual type checkers BU1 to BU4.

Input data. We run the type checkers on synthesized input expressions of varying sizes. We use balanced binary syntax trees $T(N^h, l_1 \dots l_n)$ of height h with binary inner nodes N and leaves $l_1 \dots l_n$, where $n = 2^{h-1}$. In particular, we run the type checkers for heights $h \in \{2, 4, 6, 8, 10, 12, 14, 16\}$, inner nodes $N \in \{+, \mathsf{app}\}$, and leaf sequences consisting of numeric literals $(1 \dots n)$, a single variable $(x \dots x)$, or a sequence of distinct variables $(x_1 \dots x_n)$.

We chose these trees as input to explore the impact of context requirements. Trees with numeric literals $(1 \dots n)$ as leaves are type checked under an empty typing context and yield an empty requirement set. In contrast, trees with a single variable $(x \dots x)$ as leaves are type checked under a typing context with a single entry, but co-contextual type checking introduces a distinct unification variable for each variable occurrence and has to perform lots of requirement merging yielding additional constraints. Finally, trees with a sequence of distinct variables $(x_1 \dots x_n)$ are type checked under a typing context with n entries and also yield a requirement set with n entries. In the latter case, requirement merging does not yield any additional constraints because all variables are distinct. We chose addition and application as inner nodes because they yield constraints of different complexity $\{T_1 = Num, T_2 = Num\}$ and $\{T_1 = T_2 \to U\}$, respectively.

We do not use sharing of subtrees, thus our largest trees have $2^{16} - 1 = 65535$ nodes. For comparison, the largest source file of Apache Ant 1.9.4 has 17814 Java syntax-tree nodes. In our synthesized expressions, all variables occurring in a λ-expression at the top of the generated tree are bound. Instead of type annotations, we rely on the type checkers to infer the type of bound variables. Some of the synthesized expressions are ill-typed, namely when applying a number in place of a function and when applying a variable to itself. This allows us to also evaluate the run time of failing type checks. We leave the evaluation of co-contextual type checkers on real-world programs written in modern languages like Java for future work.

Experimental setup. We measure the performance of a type checker in terms of the number of syntax-tree nodes it processes per millisecond. We use ScalaMeter[3] to measure the wall-clock run times of our Scala implementations of the above type checkers. ScalaMeter ensures proper JVM warm up, takes a sequence of measurements, eliminates outliers, and computes the mean run time of the rest. We performed the benchmark on a 3Ghz octa-core Intel Xeon E5-1680 machine with 12GB RAM, running Mac OS X 10.10.4, Scala 2.11.5 and Oracle JDK 8u5 with a fixed JVM heap size of 4GB.

Based on the mean run time and the size of the input expression tree, we calculate the nonincremental performance $\frac{\#nodes = 2^h - 1}{run\ time\ in\ ms}$ of a type checker on different inner-node and leaf-node combinations. For each combination, we report the mean performance over all height configurations as well as the speedup relative to DU. Moreover, we report the overall performance of each checker as the mean performance over

[2] The benchmarking code and raw data is available online at `https://github.com/seba--/incremental`.

[3] `http://scalameter.github.io`

Tree	DU	BU1	BU2	BU3	BU4
$T(+^h, 1 \dots n)$	1078.37	836.70 (0.78)	1164.15 (1.08)	736.81 (0.68)	1109.52 (1.03)
$T(+^h, x \dots x)$	714.74	91.37 (0.13)	267.46 (0.37)	254.92 (0.36)	241.36 (0.34)
$T(+^h, x_1 \dots x_n)$	188.27	67.77 (0.36)	218.55 (1.16)	176.66 (0.94)	211.82 (1.13)
$T(\mathsf{app}^h, 1 \dots n)$	219.27	94.56 (0.43)	302.76 (1.38)	357.14 (1.63)	294.18 (1.34)
$T(\mathsf{app}^h, x \dots x)$	185.84	27.17 (0.15)	68.38 (0.37)	127.96 (0.69)	104.09 (0.56)
$T(\mathsf{app}^h, x_1 \dots x_n)$	119.41	58.33 (0.49)	130.91 (1.10)	132.23 (1.11)	153.57 (1.29)
overall performance	417.65	195.98 (0.39)	358.70 (0.91)	297.62 (0.90)	352.42 (0.95)

(a) Nonincremental performance in nodes per millisecond (speedup relative to DU).

Tree	DU	BU1	BU2	BU3	BU4
$T(+^h, 1 \dots n)$	1507.64	n/a	37028.61 (24.56)	28532.45 (18.93)	36277.34 (24.06)
$T(+^h, x \dots x)$	1147.44	n/a	2852.65 (2.49)	9699.62 (8.45)	11512.60 (10.03)
$T(+^h, x_1 \dots x_n)$	386.18	n/a	1165.87 (3.02)	1168.82 (3.03)	1670.72 (4.33)
$T(\mathsf{app}^h, 1 \dots n)$	224.08	n/a	1564.35 (6.98)	1911.00 (8.53)	2194.19 (9.79)
$T(\mathsf{app}^h, x \dots x)$	223.05	n/a	78.55 (0.35)	795.25 (3.57)	777.94 (3.49)
$T(\mathsf{app}^h, x_1 \dots x_n)$	124.49	n/a	609.23 (4.89)	728.50 (5.85)	1178.55 (9.47)
overall performance	602.15	n/a	7216.54 (7.05)	7139.27 (8.06)	8935.22 (10.19)

(b) Incremental performance in nodes per millisecond (speedup relative to DU).

Figure 8. Nonincremental and incremental type-checking performance for PCF.

all tree shapes. For BU1, we were not able to measure the performance for $h = 14$ and $h = 16$ due to high garbage-collection overheads and consequent timeouts.

For measuring incremental performance, we fix the height of the input syntax tree at $h = 16$ (due to timeouts, we excluded BU1 from this experiment). We first perform a full initial type check. To simulate incremental changes, we invalidate all types stored for the left-most subtree of height $k \in \{2, 4, 6, 8, 10, 12, 14, 16\}$. We measure the wall-clock run times for rechecking the partially invalidated syntax trees. We calculate the mean performance of a recheck $\frac{\#nodes = 2^h - 1}{run\ time\ in\ ms}$ relative to the total size of the syntax tree. We report the mean performance over all height configurations k, the speedup relative to DU, and the overall performance as the mean performance over all tree shapes.

Nonincremental performance. Figure 8(a) shows the non-incremental performance numbers for different tree configurations with speedups relative to DU in parentheses. First, note that all type checkers except BU1 are relatively fast in that they process a syntax tree of height 16 with 65535 nodes in between 157ms and 220ms (number of nodes divided by nodes per ms). BU1 is substantially slower, especially considering that we had to abort the executions for $h = 14$ and $h = 16$.

On average, type checker DU processes 417.65 syntax-tree nodes per millisecond. BU2, BU3, and BU4 perform only slightly worse than DU. By inspecting individual rows in Figure 8(a), we see that co-contextual type checkers actually outperform DU in many cases. For example, BU4 is faster than DU when leaves are numeric expressions or distinct variables. However, all co-contextual type checkers perform comparatively bad when all leaves of the syntax refer to a single variable because a large number of requirement merging is needed. For example, in a tree of height 16 we have $2^{15} = 32768$ references to the same variable and $2^{16} - 2^{15} - 1 = 32767$ calls to $merge$, each of which generates a constraint to unify the types of the variable references. In summary, we can say that BU2 and BU4 have run-time costs similar to those of DU, but their performance varies with respect to variable occurrences.

Incremental performance. Figure 9 shows the incremental performance of DU (blue), BU2 (orange), BU3 (green), and BU4 (red) on $T(\mathsf{app}^{16}, x_1 \dots x_n)$. The x-axis shows the height of the invalidated subexpression; the y-axis shows the run time of the four type checkers. DU does not support incremental type checking and therefore takes the same time to recheck the input of size $2^{16} - 1$ independent of the size of the invalidated subexpression. In contrast, BU2, BU3, and BU4 run considerably faster. Especially for small changes, incremental type checking provides large speedups. However, the graph also reveals that the incremental type checking does not scale down to small changes lineally. Instead, we observe that incremental rechecking takes roughly the same time when invalidating a subexpression of height $k \in \{2, 4, 6, 8\}$. This is because our incremental type checkers need to traverse the whole syntax tree once in order to find the invalidated

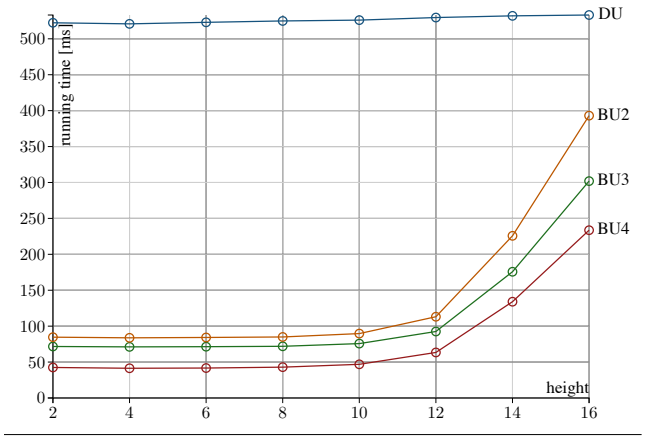


Figure 9. Incremental running time on $T(\mathsf{app}^{16}, x_1 \ldots x_n)$ for changes of size $2^{\mathsf{height}} - 1$.

subexpression. Thus, even a small change incurs the cost of traversing the syntax tree once.

Figure 8(b) presents a bigger picture of the incremental performance, where we report the mean performance over all height configurations k of the invalidated subexpression. Since DU does not support incremental type checking, it has to recheck the whole syntax tree nonincrementally for every change. The numbers for DU differ from the numbers reported in Figure 8(a) because we fixed the height to $h = 16$. For the co-contextual type checkers, we see a significant performance improvement of up to 24.56x. Incremental type checkers BU3 and BU4 also achieve good speedups when all leaves refer to the same variable, which yielded slowdowns in the nonincremental case.

The co-contextual type checkers BU2, BU3, BU4 follow different substitution strategies, where the results indicate that the choice of strategy noticeably influences performance. Deferring substitutions until the final step (BU2) and immediate substitutions (BU3) compare worse to a balanced strategy (BU4), where the type checker defers substitution until incremental constraint solving generates a certain amount of solutions.

7. Related Work

Co-contextual type rules are different from type inference, because type inference relies on the context to coordinate the types of variables [17]. That is, type inference assigns the same type to all references of the same bound variable. In contrast, co-contextual type checking assigns each variable reference a fresh type variable and coordinates them through requirement merging.

Our co-contextual formulation of type rules is related to prior work on principal typing [11, 25], not to be confused with principal types. A principal typing of some expression is a derivation that subsumes all other derivations. Specifically, a principal typing (if it exists) requires the weakest context and provides the strongest type possible for the given expression. Principal typing finds application in type inference where, similar to our work, the minimal context requirements can be inferred automatically. We extend the work on principal typing by identifying the duality between contexts and context requirements as a method for systematically constructing co-contextual type systems. Such a method has not been formulated in previous work. Moreover, prior work on principal typing only considered ad-hoc incrementalization for type checking top-level definitions, whereas we describe a method for efficient fine-grained incremental type checking.

Other formulations of type systems related to our co-contextual one have also been used in the context of compositional compilation [3] and the compositional explanation of type errors [7]. However, these systems only partially eliminated the context. In the work on compositional compilation [3], type checking within a module uses a standard contextual formulation that coordinates the types of parameters and object self-references this. For references to other modules, the type checker generates constraints that are resolved by the linker. Using our method for constructing co-contextual type systems, we can generalize this type system to eliminate the local context as well, thus enabling compositional compilation for individual methods. In the work on compositional explanations of type errors [7], only the context for monomorphic variables is eliminated, whereas a context for polymorphic variables is propagated top-down. Our extension for parametric polymorphism demonstrated that our co-contextual formulation of type rules can support polymorphic variables and eliminate the context entirely.

Snelting, Henhapl, and Bahlke's work on PSG and context relations [5, 22, 23] supports incremental analyses beyond traditional type checking and provide a join mechanism that is similar to our merge operation. However, context relations are very different from our proposal both conceptually as well as technically. Conceptually, our main conceptual finding is the duality between contexts and co-contexts that informs the design of co-contextual type checkers. For example, we used this duality to essentially derive bottom-up type checkers that support subtyping and polymorphism. In contrast, it is not obvious how to extend context relations and their unification-based approach to support subtyping (beyond enumerating subtyping for base types like int or float) or user-defined polymorphism (with explicit type application). To use context relations, the user has to come up with clever encodings using functionally augmented terms. The duality we found provides a principle for systematically identifying appropriate encodings. Technically, we do not use relations with cross-references to represent analysis results and we do not rely on a separate name resolution that runs before type checking. Instead, we use user-supplied constraint systems and context requirements. In particular, this enables us to solve constraints locally or globally and to apply solutions eagerly or propagate solutions up the tree (with performance impacts as shown in Section 6).

Kahn, Despeyroux, et al.'s work on Typol [9, 13] compiles inference rules to Prolog. Thus, Typol benefits from Prolog's support for solving predicates such as tcheck(C,E,T) for any of the variables. In contrast, our main contribution is to systematically eliminate the context through duality. In particular, the practicality of using Prolog to infer context requirements is not documented, and it is not clear if the minimal context is always found. Attali et al. present an incremental evaluator for a large class of Typol programs including type checkers [4]. However, their incrementalization requires that the context is stable and does not change. If the context changes, the previous computation is discarded and gets repeated.

Meertens describes an incremental type checker for the programming language B [15]; it collects fine-grained type requirements, but is not clear on requirement merging and also does not solve type requirements incrementally. Johnson and Walz describe a contextual type checker that incrementally normalizes type constraints while type checking a program in order to precisely identify the cause of type errors [12]. Aditya and Nikhil describe an incremental Hindley/Milner type system that only supports incremental rechecking of top-level definitions [2]. Miao and Siek describe a type checker for multi-staged programming [16]; type checking in later stages is incremental with respect to earlier stages, relying on the fact that types only get refined by staging but assumptions on the type of an expression never have to be revoked.

Wachsmuth et al. describe a task engine for name resolution and type checking [24]. After a source file changes, the incremental type checker generates new tasks for this file. The task engine reuses cached task results where possible, computes tasks that have not been seen before, and invalidates cached results that depend on tasks that have been removed. The type checker relies on an incremental name analysis, whose tasks are incrementally maintained by the same engine.

Eclipse's JDT performs incremental compilation through fine-grained dependency tracking.[4] It evolved from IBM's VisualAge [6], which stores the code and configuration of a software project in an in-memory database with separate entries for different artifacts, such as methods, variables, or classpaths. The system tracks dependencies between individual artifacts in order to update the compilation incrementally. Similarly, Facebook's Flow[5] and Hack[6] languages feature incremental type checking through a background server that watches source files in order to invalidate the type of changed files and files that are affected by them. Unfortunately, not much information on the incremental type checkers of Eclipse and Facebook is available, and it is not clear how to construct similar incremental type checkers systematically.

[4] http://eclipse.org/jdt/core/

[5] http://flowtype.org

[6] http://hacklang.org

Finally, there are general-purpose engines for incremental computations. Before implementing incremental type checkers directly, we experimented with an implementation of co-contextual type rules based on an incremental SQL-like language developed by Mitschke et al. [18]. In our experiment, the overhead of general-purpose incrementalization was significant and too large for the fine-grained incremental type checking that we are targeting. For this reason, we did not further consider other general-purpose engines for incremental computations, such as attribute grammars [8] or self-adjusting computations [1].

Our implementation strategy is similar to the general-purpose incrementalization system Adapton [10]. Like Adapton, after a program (data) modification, we also only rerun the type checker (computation) when its result is explicitly required again. This way, multiple program modifications can accumulate and be handled by a single type-check run. In contrast to Adapton, we also propagate dirty flags only on-demand.

8. Conclusions and Future Work

We presented a co-contextual formulation for type rules and described a method for systematically constructing co-contextual type systems. A co-contextual formulation of type rules avoids coordination between subderivations, which makes this formulation well-suited for parallel and incremental type checking. In this paper, we focused on incremental type checking and described a method for developing efficient incremental type checkers on top of co-contextual type systems. In particular, we applied memoization to avoid recomputing derivations for unchanged subexpressions. This enables fine-grained incremental type checking.

We presented co-contextual type systems for PCF and extensions for records, parametric polymorphism, and subtyping, and implemented corresponding incremental type checkers. Our performance evaluation shows that co-contextual type checking for PCF has performance comparable to standard contextual type checking, and incrementalization can improve performance significantly.

In future work, we want to explore parallel type checking based on a co-contextual formulation of type rules. Besides avoiding coordination for generating fresh type variables, parallelization also requires efficient strategies for distributing the syntax tree to and collecting the subresults from multiple workers while keeping the coordination overhead minimal.

Moreover, we are currently developing a co-contextual type system for Java, which involves complicated type rules and elaborate context requirements on field types and method signatures. This development will lead to a fine-grained incremental type checker for Java and will provide insights into the scalability and applicability of our approach.

Acknowledgments. We thank Klaus Ostermann, Greg Morrisett, Tillmann Rendel, Guido Salvaneschi, and the anony-

mous reviewers for helpful feedback. This work has been supported by the European Research Council, grant No. 321217.

References

[1] U. A. Acar, G. E. Blelloch, M. Blume, R. Harper, and K. Tangwongsan. An experimental analysis of self-adjusting computation. *Transactions on Programming Languages and Systems (TOPLAS)*, 32(1), 2009.

[2] S. Aditya and R. S. Nikhil. Incremental polymorphism. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 379–405. ACM, 1991.

[3] D. Ancona, F. Damiani, S. Drossopoulou, and E. Zucca. Polymorphic bytecode: Compositional compilation for Java-like languages. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 26–37. ACM, 2005.

[4] I. Attali, J. Chazarain, and S. Gilette. Incremental evaluation of natural semantics specifications. *Logical Foundations of Computer Science*, pages 87–104, 1992.

[5] R. Bahlke and G. Snelting. The PSG system: From formal language definitions to interactive programming environments. *Transactions on Programming Languages and Systems (TOPLAS)*, 8(4):547–576, 1986.

[6] L. A. Chamberland, S. F. Lymer, and A. G. Ryman. IBM VisualAge for Java. *IBM Systems Journal*, 37(3):386–408, 1998.

[7] O. Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 193–204. ACM, 2001.

[8] A. J. Demers, T. W. Reps, and T. Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 105–116. ACM, 1981.

[9] T. Despeyroux. Executable specification of static semantics. In *Semantics of Data Types*, volume 173 of *LNCS*. Springer, 1984.

[10] M. A. Hammer, Y. P. Khoo, M. Hicks, and J. S. Foster. Adapton: composable, demand-driven incremental computation. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, pages 156–166. ACM, 2014.

[11] T. Jim. What are principal typings and what are they good for? In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 42–53. ACM, 1996.

[12] G. F. Johnson and J. A. Walz. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 44–57. ACM, 1986.

[13] G. Kahn. Natural semantics. In *Proceedings of Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 22–39. Springer, 1987.

[14] A. Kuhn, G. C. Murphy, and C. A. Thompson. An exploratory study of forces and frictions affecting large-scale model-driven development. In *Proceedings of Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 7590 of *LNCS*, pages 352–367. Springer, 2012.

[15] L. G. L. T. Meertens. Incremental polymorphic type checking in B. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 265–275. ACM, 1983.

[16] W. Miao and J. G. Siek. Incremental type-checking for type-reflective metaprograms. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, pages 167–176. ACM, 2010.

[17] R. Milner. A theory of type polymorphism in programming. *Computer and System Sciences*, 17(3):348–375, 1978.

[18] R. Mitschke, S. Erdweg, M. Köhler, M. Mezini, and G. Salvaneschi. i3QL: Language-integrated live data views . In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 417–432. ACM, 2014.

[19] B. C. Pierce. *Types and programming languages*. MIT press, 2002.

[20] B. C. Pierce and D. N. Turner. Local type inference. *Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, 2000.

[21] F. Pottier. Simplifying subtyping constraints: A theory. *Information and Computation*, 170(2):153–183, 2001.

[22] G. Snelting. The calculus of context relations. *Acta Informatica*, 28(5):411–445, 1991.

[23] G. Snelting and W. Henhapl. Unification in many-sorted algebras as a device for incremental semantic analysis. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 229–235. ACM, 1986.

[24] G. Wachsmuth, G. D. P. Konat, V. A. Vergu, D. M. Groenewegen, and E. Visser. A language independent task engine for incremental name and type analysis. In *Proceedings of Conference on Software Language Engineering (SLE)*, volume 8225 of *LNCS*, pages 260–280, 2013.

[25] J. B. Wells. The essence of principal typings. In *Proceedings of International Colloquium on Automata, Languages and Programming (ICALP)*, volume 2380 of *LNCS*, pages 913–925. Springer, 2002.

A. Equivalence of Contextual and Co-Contextual PCF

In this appendix we provide the proof to Theorem 1, showing that our formulations of contextual and co-contextual PCF are equivalent.

Recall that we call a syntactic entity ground if it does not contain unification variables and we write $\Gamma \supseteq R$ if $\Gamma(x) = R(x)$ for all $x \in dom(R)$.

Lemma 1. *Let $merge(R_1, R_2) = R|_C$, $\Gamma \supseteq \sigma_1(R_1)$, $\Gamma \supseteq \sigma_2(R_2)$, and $\sigma_1(R_1)$ and $\sigma_2(R_2)$ be ground. Then $\sigma_1 \circ \sigma_2$ solves C.*

Proof. By the definition of $merge$, $C = \{R_1(x) = R_2(x) \mid x \in dom(R_1) \cap dom(R_2)\}$. Since $\Gamma \supseteq \sigma_i(R_i)$, we know $\Gamma(x) = \sigma_i(R_i(x))$ for all $x \in dom(R_i)$. In particular, $\Gamma(x) = \sigma_1(R_1(x)) = \sigma_2(R_2(x))$ for all $x \in dom(R_1) \cap dom(R_2)$. Thus, $\sigma_1 \circ \sigma_2$ solves C because $(\sigma_1 \circ \sigma_2)(R_1(x)) = \sigma_1(R_1(x)) = \sigma_2(R_2(x)) = (\sigma_1 \circ \sigma_2)(R_2(x))$ for all $x \in dom(R_1) \cap dom(R_2)$, because $\sigma_1(R_1)$ and $\sigma_2(R_2)$ are ground. □

Theorem 1 (Equivalence of contextual PCF and co-contextual PCF)**.** *A program e is typeable in contextual PCF if and only if it is typeable in co-contextual PCF:*

$\Gamma \vdash e : T \mid C$ *and* $solve(C) = \sigma$ *such that* $\sigma(T)$ *and* $\sigma(\Gamma)$ *are ground*

if and only if

$e : T' \mid C' \mid R$ *and* $solve(C') = \sigma'$ *such that* $\sigma'(T')$ *and* $\sigma'(R)$ *are ground*

If e is typeable in contextual and co-contextual PCF as above, then $\sigma(T) = \sigma'(T')$ and $\sigma(\Gamma) \supseteq \sigma'(R)$.

Proof. We first show that typeability in contextual PCF entails typeability in co-contextual PCF ($\Rightarrow$) with matching types and contexts/requirements. We proceed by structural induction on e.

- Case n with $\Gamma \vdash n : T \mid C$.
  By inversion, $T = Num$ and $C = \emptyset$.
  We choose $T' = Num$, $C' = \emptyset$, $R = \emptyset$, and $\sigma' = \emptyset$.
  Then $e : T' \mid C' \mid R$ holds, $\sigma(T) = Num = \sigma'(T')$, $\Gamma \supseteq \emptyset = \sigma'(R)$, and $\sigma'(T')$ and $\sigma'(R)$ are ground.
- Case x with $\Gamma \vdash x : T \mid C$.
  By inversion, $\Gamma(x) = T$ and $C = \emptyset$.
  We choose $T' = U$, $C' = \emptyset$, $R = \{x : U\}$, and $\sigma' = \{U \mapsto T\}$.
  Then $e : T' \mid C' \mid R$ holds, $\sigma(T) = T = \sigma'(U) = \sigma'(T')$, $\Gamma \supseteq \{x : T\} = \sigma'(R)$, $\sigma'(T')$ and $\sigma'(R)$ are ground because T is ground.
- Case $\lambda x : T_1.\ e$ with $\Gamma \vdash \lambda x : T_1.\ e : T \mid C$.
  By inversion, $x : T_1; \Gamma \vdash e : T_2 \mid C_e$, $T = T_1 \to T_2$, and $C = C_e$.
  Let $solve(C) = \sigma$.
  By IH, $e : T_2' \mid C_e' \mid R_e$ with $solve(C_e') = \sigma_e'$, $\sigma_e(T_2) = \sigma_e'(T_2')$, $(x : T_1; \Gamma) \supseteq \sigma_e'(R_e)$, $\sigma_e'(T_2')$ is ground, and $\sigma_e'(R_e)$ is ground.
  We choose $T' = T_1 \to T_2'$ and $R = R_e - x$.
  - If $x \in dom(R_e)$, then $R_e(x) = U_e$ for some U_e.
    We choose $C' = C_e' \cup \{T_1 = R_e(x)\}$ and $\sigma' = \sigma_e' \circ \{U_e \mapsto T_1\}$.
    Then $e : T' \mid C' \mid R$ holds and σ' solves C_e' and $\sigma'(T_1) = T_1 = \sigma'(U_e) = \sigma'(R_e(x))$. Moreover, $\sigma(T) = T_1 \to \sigma(T_2) = T_1 \to \sigma'(T_2') = \sigma'(T')$, $\Gamma \supset \sigma'(R_e - x)$, $\sigma'(T')$ is ground because T_1 and $\sigma'(T_2')$ are ground, and $\sigma'(R)$ is ground because $\sigma_e'(R_e)$ is ground.
  - If $x \notin dom(R_e)$, we choose $C' = C_e'$ and $\sigma' = \sigma_e'$.
    Then $e : T' \mid C' \mid R$ holds, σ' solves C', $\sigma(T) = T_1 \to \sigma(T_2) = T_1 \to \sigma'(T_2') = \sigma'(T')$, $\Gamma \supset \sigma'(R_e - x)$, $\sigma'(T')$ is ground because T_1 and $\sigma'(T_2')$ are ground, and $\sigma'(R)$ is ground because $\sigma_e'(R_e)$ is ground.
- Case $e_1\ e_2$ with $\Gamma \vdash e_1\ e_2 : T \mid C$.
  By inversion, $\Gamma \vdash e_1 : T_1 \mid C_1$, $\Gamma : e_2 \mid T_2 \mid C_2$, $T = U$, and $C = C_1 \cup C_2 \cup \{T_1 = T_2 \to U\}$.
  Let $solve(C) = \sigma$, which also solves C_1, C_2, and $T_1 = T_2 \to U$ such that $\sigma(U)$ is ground.
  By IH for $i \in \{1, 2\}$, $e_i : T_i' \mid C_i' \mid R_i$ with $solve(C_i') = \sigma_i'$, $\sigma(T_i) = \sigma'(T_i')$, $\Gamma \supseteq \sigma_i'(R_i)$, $\sigma_i'(T_i')$ is ground, and $\sigma_i'(R_i)$ is ground.
  Let $merge(R_1, R_2) = R|_{C_r'}$. We choose $T' = U$, $C' = C_1' \cup C_2' \cup \{T_1' = T_2' \to U\} \cup C_r'$, and $\sigma' = \sigma_1' \circ \sigma_2' \circ \{U \mapsto \sigma(U)\}$.
  Then $e_1\ e_2 : T' \mid C' \mid R$ holds and σ' solves C' because it solves C_1', C_2', C_r' (by Lemma 1), and $\sigma'(T_1') = \sigma(T_1) = \sigma(T_2) \to \sigma(U) = \sigma'(T_2') \to \sigma'(U) = \sigma'(T_2' \to U)$.
  Moreover, $\sigma'(T') = \sigma'(U) = \sigma(U) = \sigma(T)$, $\Gamma \supseteq \sigma'(R_1) \cup \sigma'(R_2) \supseteq \sigma'(R)$ by the definition of $merge$, $\sigma'(T')$ is ground because $\sigma(U)$ is ground, and $\sigma'(R)$ is ground because $\sigma_i'(R_i)$ are ground.
- Case $e_1 + e_2$ with $\Gamma \vdash e_1 + e_2 : T \mid C$.
  By inversion, $\Gamma \vdash e_1 : T_1 \mid C_1$, $\Gamma : e_2 \mid T_2 \mid C_2$, $T = Num$, and $C = C_1 \cup C_2 \cup \{T_1 = Num, T_2 = Num\}$.
  Let $solve(C) = \sigma$, which also solves C_1, C_2,$T_1 = Num$, and $T_2 = Num$.
  By IH for $i \in \{1, 2\}$, $e_i : T_i' \mid C_i' \mid R_i$ with $solve(C_i') = \sigma_i'$, $\sigma_i(T_i) = \sigma'(T_i')$, $\Gamma \supseteq \sigma'(R_i)$, $\sigma'(T_i')$ is ground, and $\sigma'(R_i)$ is ground.
  Let $merge(R_1, R_2) = R|_{C_r'}$. We choose $T' = Num$, $C' = C_1' \cup C_2' \cup \{T_1' = Num, T_2' = Num\} \cup C_r'$, and $\sigma' = \sigma_1' \circ \sigma_2'$.
  Then $e_1 + e_2 : T' \mid C' \mid R$ and σ' solves C' because it solves C_1', C_2', C_r' (by Lemma 1), and $\sigma'(T_i') = \sigma(T_i) = Num$.
  Moreover, $\sigma'(T') = Num = \sigma(T)$, $\Gamma \supseteq \sigma'(R_1) \cup \sigma'(R_2) \supseteq \sigma'(R)$ by the definition of $merge$, $\sigma'(T') =$

Num is ground, and $\sigma'(R)$ is ground because $\sigma'_i(R_i)$ are ground.

- Case *if0* $e_1\ e_2\ e_3$ with $\Gamma \vdash$ *if0* $e_1\ e_2\ e_3 : T \mid C$.
  By inversion for $i \in \{1,2,3\}$, $\Gamma \vdash e_i : T_i \mid C_i$, $T = T_2$, and$C = C_1 \cup C_2 \cup C_3 \cup \{T_1 = Num, T_2 = T_3\}$.
  Let $solve(C) = \sigma$, which also solves C_1, C_2, C_3, $T_1 = Num$, and $T_2 = T_3$.
  By IH, $e_i : T'_i \mid C'_i \mid R_i$ with $solve(C'_i) = \sigma'_i$, $\sigma'_i(T'_i) = \sigma(T_i)$, $\Gamma \supseteq \sigma'_i(R_i)$, $\sigma'(T'_i)$ is ground, and $\sigma'(R_i)$ is ground.
  Let $merge(R_2, R_3) = R_{2,3}|_{C'_{2,3}}$, $merge(R_1, R_{2,3}) = R_{1,2,3}|_{C'_{1,2,3}}$. We choose $R = R_{1,2,3}$, $T' = T'_2$, $C' = C'_1 \cup C'_2 \cup C'_3 \cup \{T'_1 = Num, T'_2 = T'_3\} \cup C'_{2,3} \cup C'_{1,2,3}$, and $\sigma = \sigma_1 \circ \sigma_2 \circ \sigma_3$.
  Then *if0* $e_1\ e_2\ e_3 : T' \mid C' \mid R$ and σ' solves C' because it solves C'_1, C'_2,C'_3, $C'_{2,3}$ (by Lemma 1), $C'_{1,2,3}$ (by Lemma 1), and $\sigma'(T'_1) = \sigma(T_1) = Num$ as well as $\sigma'(T'_2) = \sigma(T_2) = \sigma(T_3) = \sigma'(T'_3)$.
  Moreover, $\sigma'(T') = \sigma'(T'_2) = \sigma(T_2) = \sigma(T)$, $\Gamma \supseteq \sigma'(R_1) \cup \sigma'(R_2) \cup \sigma'(R_3) \supseteq \sigma'(R_1) \cup \sigma'(R_{2,3}) \supseteq \sigma'(R_{1,2,3})$ by the definition of $merge$, $\sigma'(T')$ is ground because $\sigma'(T'_2)$ is ground, and $\sigma'(R)$ is ground because $\sigma'_i(R_i)$ are ground.

- Case *fix* e with $\Gamma \vdash$ *fix* $e : T \mid C$.
  By inversion, $\Gamma \vdash e : T_e \mid C_e$, $T = U$, and $C = C_e \cup \{T_e = U \to U\}$.
  Let $solve(C) = \sigma$, which solves C_e and $T_e = U \to U$ such that $\sigma(U)$ is ground.
  By IH, $e : T'_e \mid C'_e \mid R_e$ with $solve(C'_e) = \sigma'_e$, $\sigma'_e(T'_e) = \sigma(T_e)$, $\Gamma_e \supseteq \sigma'(R_e)$, $\sigma'(T'_e)$ is ground, and $\sigma'(R_e)$ is ground.
  We choose $R = R_e$, $T' = U$, $C' = C'_e \cup \{T'_e = U \to U\}$, and $\sigma' = \sigma'_e \circ \{U \mapsto \sigma(U)\}$.
  Then *fix* $e : T' \mid C' \mid R$ and σ' solves C' because it solves C'_e and $\sigma'(T'_e) = \sigma(T_e) = \sigma(U \to U) = \sigma(U) \to \sigma(U) = \sigma'(U) \to \sigma'(U) = \sigma'(U \to U)$.
  Moreover, $\sigma'(T') = \sigma'(U) = \sigma(U) = \sigma(T)$, $\Gamma = \Gamma_e \supseteq \sigma'(R_e) = \sigma'(R)$, $\sigma'(T')$ is ground because $\sigma(U)$ ground, and $\sigma'(R)$ is ground because $\sigma'(R_e)$ is ground.

Next we show that typeability in co-contextual PCF entails typeability in contextual PCF ($\Leftarrow$) with matching types and contexts/requirements. We again proceed by structural induction on e.

- Case n with $n : T' \mid C' \mid R$.
  By inversion, $T' = Num$, $C' = \emptyset$, and $R = \emptyset$.
  Let $solve(C') = \sigma'$. We choose $\Gamma = \emptyset$, $T = Num$, $C = \emptyset$, $\sigma = \emptyset$.
  Then $\Gamma \vdash e : T \mid C$ holds, $\sigma(T) = Num = \sigma'(T')$, $\Gamma = \emptyset = \sigma'(R)$, and $\sigma(T)$ is ground.

- Case x with $x : T' \mid C' \mid R$.
  By inversion, $T' = U$, $C' = \emptyset$, and $R = \{x : U\}$.
  Let $\sigma'(U) = T_x$ for some T_x that we know is ground.
  We choose $\Gamma = x : T_x; \emptyset$, $T = T_x$, $C = \emptyset$, and $\sigma = \emptyset$.
  Then $\Gamma \vdash e : T \mid C$ holds, $\sigma(T) = T_x = \sigma'(U)$,$\Gamma = (x : T_x; \emptyset) \supseteq \{x : T_x\} = \sigma'(R)$, and $\sigma(T)$ is ground because T_x is ground.

- Case $\lambda x : T_1.\ e$ with $\lambda x : T_1.\ e : T' \mid C' \mid R$.
  By inversion, $e : T'_2 \mid C'_e \mid R_e$, $T' = T_1 \to T'_2$, and $R = R_e - x$ for some T'_2, C'_e, and R_e.
  Let $solve(C') = \sigma'$, which also solves C'_e.
  By IH, $\Gamma_e \vdash T_2 : e \mid C_e$ with $solve(C_e) = \sigma_e$, $\sigma_e(T_2) = \sigma'(T'_2)$, $\Gamma_e \supseteq \sigma'(R_e)$, and $\sigma_e(T_2)$ is ground. The latter entails $\Gamma_e(x) = \sigma'(R_e(x))$ for all $x \in dom(R_e)$.
  We choose $T = T_1 \to T_2$, $C = C_e$, and $\sigma = \sigma_e$ such that $\sigma(T) = T_1 \to \sigma(T_2) = T_1 \to \sigma'(T'_2) = \sigma'(T')$ and $\sigma(T)$ is ground because T_1 and $\sigma_e(T_2)$ are ground.
  - If $x \in dom(R_e)$, then $(T_1 = R_e(x)) \in C'$ and $T_1 = \sigma'(R_e(x)) = \Gamma_e(x)$. Thus, by swapping $\Gamma_e = (x : T_1; \Gamma)$ for some Γ such that $\Gamma \vdash e : T \mid C$ holds, σ solves C, and $\Gamma \supseteq \Gamma_e - x \supseteq \sigma'(R_e) - x = \sigma'(R_e - x)$.
  - If $x \notin dom(R_e)$, the x is not free in e and we get $x : T_1; (\Gamma_e - x) \vdash T_2 : e \mid C_e$ by strengthening and weakening. We choose $\Gamma = \Gamma_e - x$ such that (i) holds and $\Gamma = \Gamma_e - x \supseteq \sigma'(R_e) - x = \sigma'(R_e - x)$.

- Case $e_1\ e_2$ with $e_1\ e_2 : T' \mid C' \mid R$.
  By inversion, $e_1 : T'_1 \mid C'_1 \mid R_1$, $e_2 : T'_2 \mid C'_2 \mid R_2$, $merge(R_1, R_2) = R|_{C'_r}$, $T' = U$, and $C' = C'_1 \cup C'_2 \cup \{T'_1 = T'_2 \to U\} \cup C'$.
  Let $solve(C') = \sigma'$, which also solves C'_1, C'_2, C'_r, and $T'_1 = T'_2 \to U$ such that $\sigma'(U)$ is ground.
  By IH for $i \in \{1,2\}$, $\Gamma_i \vdash e_i : T_i \mid C_i$ with $solve(C_i) = \sigma_i$, $\sigma_i(T_i) = \sigma'(T'_i)$, $\Gamma_i \supseteq \sigma'(R_i)$, and $\sigma_i(T_i)$ is ground.
  We choose $\Gamma = \Gamma_1; \Gamma_2$, $T = U$, $C = C_1 \cup C_2 \cup \{T_1 = T_2 \to U\}$, and $\sigma = \sigma_1 \circ \sigma_2 \circ \{U \mapsto \sigma'(U)\}$.
  Since $\sigma_i(\Gamma_i) \supseteq \sigma'(R_i)$, Γ only extends Γ_1 and Γ_2 with variables that are not free in e_1 and e_2, respectively.Thus, $\Gamma \vdash e_i : T_i \mid C_i$ and $\Gamma \vdash e_1\ e_2 : U \mid C$. σ solves C because it solves C_1 and C_2 and $\sigma(T_1) = \sigma'(T'_1) = \sigma'(T'_2 \to U) = \sigma'(T'_2) \to \sigma'(U) = \sigma(T_2) \to \sigma(U) = \sigma(T_2 \to U)$.
  Moreover, $\sigma(T) = \sigma(U) = \sigma'(U) = \sigma'(T')$, $\Gamma = \Gamma_1; \Gamma_2 \supseteq \sigma'(R_1) \cup \sigma'(R_2) \supseteq \sigma'(R)$ by the definition of $merge$, and $\sigma(T)$ is ground because $\sigma'(U)$ is ground.

- Case $e_1 + e_2$ with $e_1 + e_2 : T' \mid C' \mid R$.
  By inversion for $i \in \{1,2\}$, $e_i : T'_i \mid C'_i \mid R_i$, $merge(R_1, R_2) = R|_{C'_r}$, $T' = Num$, and $C' = C'_1 \cup C'_2 \cup \{T'_1 = Num, T'_2 = Num\} \cup C'_r$.
  Let $solve(C') = \sigma'$, which also solves C'_1, C'_2, C'_r, $T'_1 = Num$, and $T'_2 = Num$.
  By IH, $\Gamma_i \vdash e_i : T_i \mid C_i$ with $solve(C_i) = \sigma_i$, $\sigma_i(T_i) = \sigma'(T'_i)$, $\Gamma_i \supseteq \sigma'(R_i)$, and $\sigma_i(T_i)$ is ground.
  We choose $\Gamma = \Gamma_1; \Gamma_2$, $T = Num$, $C = C_1 \cup C_2 \cup \{T_1 = Num, T_2 = Num\}$, and $\sigma = \sigma_1 \circ \sigma_2$.

Since $\sigma_i(\Gamma_i) \supseteq \sigma'(R_i)$, Γ only extends Γ_1 and Γ_2 with variables that are not free in e_1 and e_2, respectively.Thus, $\Gamma \vdash e_i : T_i \mid C_i$ and $\Gamma \vdash e_1 + e_2 : \mathit{Num} \mid C$.
σ solves C because it solves C_1 and C_2 and $\sigma(T_i) = \sigma'(T_i') = \mathit{Num}$.
Moreover, $\sigma(T) = \mathit{Num} = \sigma'(T')$, $\Gamma = \Gamma_1;\Gamma_2 \supseteq \sigma'(R_1) \cup \sigma'(R_2) \supseteq \sigma'(R)$ by the definition of merge, and $\sigma(T) = \mathit{Num}$ is ground.

- Case *if0* $e_1\ e_2\ e_3$ with *if0* $e_1\ e_2\ e_3 : T' \mid C' \mid R$.
By inversion for $i \in \{1,2,3\}$, $e_i : T_i' \mid C_i' \mid R_i$, $\mathit{merge}(R_2, R_3) = R_{2,3}|_{C_{2,3}}$, $\mathit{merge}(R_1, R_{2,3}) = R_{1,2,3}|_{C_{1,2,3}}$, $T' = T$, $R = R_{1,2,3}$, and $C' = C_1' \cup C_2' \cup C_3' \cup \{T_1' = \mathit{Num}, T_2' = T_3'\} \cup C_{2,3} \cup C_{1,2,3}$.
Let $\mathit{solve}(C') = \sigma'$, which also solves C_1', C_2', C_3', $C_{2,3}$, $C_{1,2,3}$, $T_1' = \mathit{Num}$, and $T_2' = T_3'$.
By IH, $\Gamma_i \vdash e_i : T_i \mid C_i$ with $\mathit{solve}(C_i) = \sigma_i$, $\sigma_i(T_i) = \sigma'(T_i')$, $\Gamma_i \supseteq \sigma'(R_i)$, and $\sigma_i(T_i)$ is ground.
We choose $\Gamma = \Gamma_1;\Gamma_2;\Gamma_3$, $T = T_2$, $C = C_1 \cup C_2 \cup C_3 \cup \{T_1 = \mathit{Num}, T_2 = T_3\}$, and $\sigma = \sigma_1 \circ \sigma_2 \circ \sigma_3$.
Since $\sigma_i(\Gamma_i) \supseteq \sigma'(R_i)$, Γ only extends Γ_1, Γ_2, and Γ_3 with variables that are not free in e_1, e_2, and e_3, respectively.Thus, $\Gamma \vdash e_i : T_i \mid C_i$ and $\Gamma \vdash \mathit{if0}\ e_1\ e_2\ e_3 : T_2 \mid C$.
σ solves C because it solves C_1, C_2, and C_3 and $\sigma(T_1) = \sigma'(T_1') = \mathit{Num}$ as well as $\sigma(T_2) = \sigma'(T_2') = \sigma'(T_3') = \sigma(T_3)$.
Moreover, $\sigma(T) = \sigma_2(T_2) = \sigma'(T_2') = \sigma'(T')$, $\Gamma = \Gamma_1;\Gamma_2;\Gamma_3 \supseteq \sigma'(R_1) \cup \sigma'(R_2) \cup \sigma'(R_3) \supseteq \sigma'(R_1) \cup \sigma'(R_{2,3}) \supseteq \sigma'(R_{1,2,3})$ by the definition of merge, and $\sigma(T)$ is ground because $\sigma_2(T_2)$ is ground.

- Case *fix* e with *fix* $e : T' \mid C' \mid R$.
By inversion, $e : T_e' \mid C_e' \mid R_e$, $T' = U$, $C' = C_e' \cup \{T_e' = U \to U\}$, and $R = R_e$.
Let $\mathit{solve}(C') = \sigma'$, which solves C_e' and $T_e' = U \to U$ such that $\sigma'(U)$ is ground.
By IH, $\Gamma_e \vdash e : T_e \mid C_e$ with $\mathit{solve}(C_e) = \sigma_e$, $\sigma_e(T_e) = \sigma'(T_e')$, $\Gamma_e \supseteq \sigma'(R_e)$, and $\sigma(T_e)$ is ground.
We choose $\Gamma = \Gamma_e$, $T = U$, $C = C_e \cup \{T_e = U \to U\}$, and $\sigma = \sigma_e \circ \{U \mapsto \sigma'(U)\}$.
Then $\Gamma \vdash \mathit{fix}\ e : T \mid C$ and σ solves C because it solves C_e and $\sigma(T_e) = \sigma'(T_e') = \sigma'(U \to U) = \sigma'(U) \to \sigma'(U) = \sigma(U) \to \sigma(U) = \sigma(U \to U)$. Moreover, $\sigma(T) = \sigma(U) = \sigma'(U) = \sigma'(T')$, $\Gamma = \Gamma_e \supseteq \sigma'(R_e) = \sigma'(R)$, and $\sigma(T)$ is ground because $\sigma'(U)$ is ground.

□