Incremental Bidirectional Typing via Order Maintenance

THOMAS J. PORTER, University of Michigan, USA
MARISA KIRISAME, University of Utah, USA
IVAN WEI, University of Michigan, USA
PAVEL PANCHEKHA, University of Utah, USA
CYRUS OMAR, University of Michigan, USA

Live programming environments provide various semantic services, including type checking and evaluation, continuously as the user is editing the program. The live paradigm promises to improve the developer experience, but liveness is an implementation challenge, particularly when working with large programs. This paper specifies and efficiently implements a system that is able to incrementally update type information for a live program in response to fine-grained program edits. This information includes type error marks and information about the expected and actual type of every expression. The system is specified type-theoretically as a small-step dynamics that propagates updates through the marked and annotated program. Most updates flow according to a base bidirectional type system. Additional pointers are maintained to connect bound variables to their binding locations, with type updates traversing these pointers directly. Order maintenance data structures are employed to efficiently maintain these pointers and to prioritize the order of update propagation. We prove this system is equivalent to naive reanalysis in the Agda theorem prover, along with other important metatheoretic properties. We then provide an efficient OCaml implementation, detailing a number of impactful optimizations. We evaluate this implementation's performance with a large stress-test and find that it is able to achieve multiple orders of magnitude speed-up compared to from-scratch reanalysis.

CCS Concepts: • **Software and its engineering** → *Development frameworks and environments*; *Incremental compilers*.

Additional Key Words and Phrases: incremental type checking, bidirectional typing, order maintenance

ACM Reference Format:
Thomas J. Porter, Marisa Kirisame, Ivan Wei, Pavel Panchekha, and Cyrus Omar. 2025. Incremental Bidirectional Typing via Order Maintenance. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 339 (October 2025), 28 pages. https://doi.org/10.1145/3763117

1 Introduction

Traditional programming language implementations are designed for batch usage with complete programs, so they often struggle to keep up with the demands of live programming environments, which aim to provide *live* (i.e. continuously available) semantic feedback to the programmer throughout the editing process, even when the program is incomplete or contains localized errors [29].

This paper focuses on the problem of providing live *type information* while a program sketch (i.e. a program with syntactic *holes* in various positions) is being edited. By type information, we mean the locations and causes of type errors as well as information about the expected and actual type at *every* location in the program sketch, even in the presence of type errors at other locations. In order to achieve liveness, our central constraint is that the system must not need to traverse

Authors' Contact Information: Thomas J. Porter, University of Michigan, USA; Marisa Kirisame, University of Utah, USA; Ivan Wei, University of Michigan, USA; Pavel Panchekha, University of Utah, USA; Cyrus Omar, University of Michigan, USA.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2025 Copyright held by the owner/author(s).
ACM 2475-1421/2025/10-ART339
https://doi.org/10.1145/3763117

the entire program sketch (which we assume to be arbitrarily large) between edits. Instead, the system should be able to *incrementally* update the collected type information, at a computational cost proportional to the number of potentially affected locations.

Even this essential computational cost may be inherently high for some changes to large programs, so we go further and aim to minimize situations where editing is blocked waiting for updates to the type information to propagate, and eliminate situations where updates caused by previous edits are rolled back and recomputed when a subsequent edit is performed. Instead, we allow updates to propagate at finite speed through the program sketch, even as new edits come in, with correctness guaranteed once this interleaving of edits and updates has fully quiesced.

These uncompromising technical aims are motivated both by the increasing prevalence of multi-million-line code bases (often organized into "monorepos") in many organizations, and by a vision of the future of programming in which large-scale scientific and social collaborations occur within a shared, live programming environment that allows thousands of participants to collaboratively edit a single "planetary-scale" live program [5, 18].

In pursuit of these aims, this paper develops a foundational type-theoretic calculus of incremental type information maintenance for a bidirectional type system (i.e. a type system organized around local type inference) [11, 31]. We equip this calculus with a comprehensive metatheory mechanized in the Agda proof assistant. Based on this core calculus, we develop an implementation that employs techniques pioneered in web browsers for incremental page layout computations [19]—most centrally, the use of order maintenance data structures [6]—to realize the promised speedups.

We start in Section 2 by reviewing the necessary background on bidirectional typing [11, 31], gradual typing [39, 40], and the *marked lambda calculus (MLC)* [50], which specifies a total procedure for bidirectional type error marking. In Section 3 we develop a variant of this calculus, called the *marked and annotated lambda calculus (MALC)*, which simplifies the specification of error marking and adds additional annotations recording the expected and inferred types at each location in the program. Collectively, we call error marks and type annotations *type information*. We also introduce a calculus of purely syntactic structural edit actions on MALC terms. We expect that these tree-structured edit actions will be generated either directly by a structure editor like Hazel [28], Scratch [22], or Pantograph [33], or else that they will be inferred by a parser together with a tree differencing algorithm [8]; incrementalizing these systems is beyond the scope of this paper.

We then proceed to the central problem confronted by this paper: incrementalizing the static semantics of MALC in response to these structural edits. We start with an overview of Incremental MALC by example in Section 4 and then fully specify its semantics and metatheory in Section 5. The key idea is to specify the semantics of type information updates as a small-step *update propagation* dynamics. During update propagation, the calculus maintains an *update propagation frontier* through the use of dirty bits on the types within MALC terms. Each update propagation step considers a dirty type at the frontier, calculates its local ramifications, and propagates the frontier correspondingly. This process is kicked off by Incremental MALC's edit action semantics, which creates initial dirty bits at the location of the edit, and critically, at other locations implied by the binding structure of the program. We show that edit actions and updates can be interleaved confluently. Consequently, editing is only blocked for the duration of individual edits and propagation steps, which are generally very short. We establish that correctness (with respect to MALC's semantics) is ensured once update propagation quiesces, i.e. when the frontier is empty.

Incremental MALC specifies certain critical operations related to bindings declaratively, in the usual type-theoretic style, and naive implementations of these operations would require substantial subtree traversals. We turn in Section 6 to developing the data structures and algorithms necessary to efficiently implement Incremental MALC. We call our implementation Malcom, a portmanteau of MALC and "order maintenance" due to the central role of efficient order maintenance data structures in the implementation. We then evaluate the performance of Malcom in Section 7, comparing the incremental implementation to the from-scratch implementation. We find that for partially randomized edits to a large synthetic functional program designed to serve as a stress-test, a 275.96× speedup is obtained relative to the same system in from-scratch mode. We conclude with a review of related work in Section 8 and discussion and directions for future research in Section 9.

2 Background

The type system considered in this paper is a variant of the marked lambda calculus (MLC), introduced recently by Zhao et al. [50]. The marked lambda calculus combines **bidirectional typing** and **gradual typing** to achieve "total type error localization and recovery," that is, the ability to localize type errors occurring throughout a program, without the presence of one error preventing the localization of another. The key components of this approach are summarized below.

Bidirectional Typing. Bidirectional typing is an approach to type checking that splits the ordinary typing judgement into two judgements: an analytic judgment $\Gamma \vdash e \Leftarrow \tau$ for checking whether the expression e has type τ under context Γ, and a synthetic judgment $\Gamma \vdash e \Rightarrow \tau$, which infers a type from e [11, 31]. The inference rules for the analytic and synthetic judgments directly specify a typing algorithm, and because of the locality of the flow of typing information, errors are naturally localized to the AST node where analysis or synthesis fails, unlike with unification-based inference. Zhao et al. [50] also give a unification-based type inference system on top of the bidirectionally typed core, but we do not consider this extension in this paper.

Gradual Typing. Gradual typing allows type checking intermixed typed and untyped code. The gradually typed lambda calculus extends the simply typed lambda calculus with the unknown (or dynamic) type, ?, and replaces type equality with a type consistency relation [39, 40]. The unknown type is consistent with every type. In live programming environments, gradual typing allows the type checker to handle type-incorrect programs that occur as the user edits.

Error Marks. The marked lambda calculus centers around the *marking* procedure, a total function that transforms an ordinary expression e into a marked expression $\check{e}$ by inserting error marks. Error marks indicate the presence and nature of a type error localized to an expression. Intuitively, error marks are formal representations of the "red squiggles" one might see in a program editor.

The marking procedure is defined bidirectionally, with an analytic marking judgment $\Gamma \vdash e \rightsquigarrow \check{e} \Leftarrow \tau$ and a synthetic marking judgment $\Gamma \vdash e \rightsquigarrow \check{e} \Rightarrow \tau$. These have the same interpretations as the ordinary bidirectional judgments, but with the additional output $\check{e}$ (pronounced "e check") and the additional property of specifying total functions. Any input program e, no matter how ill-typed, can be marked in either mode. The key to this totality is the use of the gradual type to recover from localized errors. Where a type error would occur in a simple type system, a term is assigned the unknown type in the marked lambda calculus. This allows typing to proceed optimistically, as the unknown type is consistent with all types and will not introduce additional downstream errors. The following example shows the case of a number literal applied to a free variable; the number is marked for synthesizing a non-function type while in function position, the variable is marked as free, and the program synthesizes the unknown type. Here, $\llparenthesis e \rrparenthesis$ is a marked expression and the subscripts and superscripts are simply symbolic representations of the "error message".

$$\varnothing \vdash 1\ x \rightsquigarrow \llparenthesis 1 \rrparenthesis^{\Rightarrow}_{\blacktriangleright \not\rightarrow}\ \llparenthesis x \rrparenthesis_{\square} \Rightarrow\ ?$$

3 The Marked and Annotated Lambda Calculus

We now introduce the marked and annotated lambda calculus (MALC), which modifies MLC in two ways in support of the goals of this paper. First, in MALC, terms store boolean marks indicating

x	$\in$	Variable	
$x?$	$\in$	Binding	$? \mid x$
τ	$\in$	Type	$? \mid \mathsf{b} \mid \tau \to \tau$
σ	$\in$	TypeOpt	$\square \mid \tau$
e	$\in$	BareExp	$? \mid \mathsf{c} \mid x \mid \lambda(x? : \tau).\, e \mid e \lhd e \mid e : \tau$
m	$\in$	Mark	$\checkmark \mid \times$
$\check{e}^{\Rightarrow}$	$\in$	MarkedSynExp	$\dot{\check{e}} \overset{\sigma}{\Rightarrow}$
$\dot{\check{e}}$	$\in$	MarkedConExp	$? \mid \mathsf{c} \mid x_m \mid \lambda(x? : \tau)_{m,m}.\, {}^{\Rightarrow}\check{e} \mid {}^{\Rightarrow}\check{e} \lhd_m {}^{\Rightarrow}\check{e} \mid {}^{\Rightarrow}\check{e} : \tau$
${}^{\Rightarrow}\check{e}$	$\in$	MarkedAnaExp	$\overset{\sigma}{\Rightarrow}_m \check{e}^{\Rightarrow}$
$\check{p}$	$\in$	MarkedProgram $\subseteq$ MarkedAnaExp	$\overset{\square}{\Rightarrow}_{\checkmark} \check{e}^{\Rightarrow}$
Γ	$\in$	Context	$\emptyset \mid \Gamma,\ x : \tau$

Fig. 1. Syntax of MALC

whether an error has been localized to that term, rather than marks being represented as term constructors. Second, every term is annotated with optional analyzed and synthesized types.

3.1 Syntax

The syntax of MALC is given in Figure 1. Types, τ, are drawn from a standard simple type system, with the addition of the unknown type of gradual type theory, ?, which also serves as a type hole. Bare expressions, e, are the ordinary expression terms in a simply typed language, consisting of constants, variables, function abstractions, function applications, and type ascriptions. Our notation for applications is nonstandard, using $\lhd$ to provide a syntactic anchor for the mark on this form. Bare expressions also allow for expression holes, ?, representing the incomplete parts of a program that occur as a user edits a program. We likewise allow the binding variable, $x?$, in a lambda abstraction to be a "binding hole," which binds nothing, to support flexible editing.

Marked expressions are like bare expressions but with additional type information. Marked expressions are defined by three mutually recursive sorts: marked synthetic expressions, $\check{e}^{\Rightarrow}$, marked constructor expressions, $\dot{\check{e}}$, and marked analytic expressions, ${}^{\Rightarrow}\check{e}$. The marked synthetic expression constructor stores an optional synthesized type, σ; the marked constructor expression constructors are the core syntactic forms; and the marked analytic expression constructor stores an optional analyzed type. The arrows suggest a flow of information from left to right: analyzed types come from the left, and synthesized types proceed to the right. Marked analytic expressions also include a consistency mark, m, which indicates consistency between the analyzed and synthesized types: $\times$ indicates the presence of a error and $\checkmark$ indicates the absence of one.

Marked programs, $\check{p}$, are modeled as analytic expressions with no analyzed type. They are not just synthetic expressions because the incremental version of the calculus makes use of the empty analytic data at the root of the program.

In addition to the consistency marks, marked constructor expressions also have marks. Each mark position corresponds to a possible kind of static error. Variables have a mark indicating whether they are free. Function abstractions have two marks, the first corresponding to whether the abstraction is analyzed against a non-function type, and the second corresponding to whether the domain of the analyzed type is inconsistent with the annotation on the abstraction. Function applications have a mark indicating whether the first child synthesizes a non-function type.

The formal syntax is quite elaborate, but most of it would be hidden from the user by default, with error marks only displayed if set to $\times$ and local types only displayed when queried.

$$\boxed{\Gamma \vdash e \rightsquigarrow \check{e}^{\Rightarrow}}$$

$$\textsc{MarkHole}\ \frac{}{\Gamma \vdash\ ?\rightsquigarrow\ ?\overset{?}{\Rightarrow}} \qquad \textsc{MarkConst}\ \frac{}{\Gamma \vdash\ \mathrm{c} \rightsquigarrow\ \mathrm{c}\overset{\mathrm{b}}{\Rightarrow}} \qquad \textsc{MarkVar}\ \frac{x_m : \tau \in \Gamma}{\Gamma \vdash x \rightsquigarrow x_m\overset{\tau}{\Rightarrow}}$$

$$\textsc{MarkAsc}\ \frac{\Gamma \vdash \tau \Rightarrow e \rightsquigarrow {}^{\Rightarrow}\check{e}}{\Gamma \vdash e : \tau \rightsquigarrow {}^{\Rightarrow}\check{e} : \tau \overset{\tau}{\Rightarrow}} \qquad \textsc{MarkSynFun}\ \frac{\Gamma,\ x? : \tau_1 \vdash e \rightsquigarrow \dot{\check{e}}\overset{\tau_2}{\Rightarrow}}{\Gamma \vdash \lambda(x? : \tau_1).\, e \rightsquigarrow \lambda(x? : \tau_1)_{\checkmark,\checkmark}.\left(\overset{\square}{\Rightarrow}_{\checkmark}\dot{\check{e}}\overset{\tau_2}{\Rightarrow}\right)\overset{\tau_1\to\tau_2}{\Rightarrow}}$$

$$\textsc{MarkAp}\ \frac{\Gamma \vdash e_1 \rightsquigarrow \dot{\check{e}}\overset{\tau}{\Rightarrow} \qquad \tau \blacktriangleright_{\vec{m}} \tau_1 \to \tau_2 \qquad \Gamma \vdash \tau_1 \Rightarrow e_2 \rightsquigarrow {}^{\Rightarrow}\check{e}}{\Gamma \vdash e_1 \lhd e_2 \rightsquigarrow \left(\overset{\square}{\Rightarrow}_{\checkmark}\dot{\check{e}}\overset{\tau}{\Rightarrow}\right) \lhd_m {}^{\Rightarrow}\check{e}\overset{\tau_2}{\Rightarrow}}$$

$$\boxed{\Gamma \vdash \tau \Rightarrow e \rightsquigarrow \check{e}^{\Rightarrow}}$$

$$\textsc{MarkSubsume}\ \frac{e\ \text{subsumable} \qquad \Gamma \vdash e \rightsquigarrow \dot{\check{e}}\overset{\tau_2}{\Rightarrow} \qquad \tau_1 \sim_m \tau_2}{\Gamma \vdash \tau_1 \Rightarrow e \rightsquigarrow \overset{\tau_1}{\Rightarrow}_m\dot{\check{e}}\overset{\tau_2}{\Rightarrow}}$$

$$\textsc{MarkAnaFun}\ \frac{\tau \blacktriangleright_{\vec{m}_1} \tau_2 \to \tau_3 \qquad \tau_1 \sim_{m_2} \tau_2 \qquad \Gamma,\ x? : \tau_1 \vdash \tau_3 \Rightarrow e \rightsquigarrow {}^{\Rightarrow}\check{e}}{\Gamma \vdash \tau \Rightarrow \lambda(x? : \tau_1).\, e \rightsquigarrow \overset{\tau}{\Rightarrow}_{\checkmark}\lambda(x? : \tau_1)_{m_1,m_2}.{}^{\Rightarrow}\check{e}\overset{\square}{\Rightarrow}}$$

$$\boxed{e \rightsquigarrow \check{p}}$$

$$\textsc{MarkProgram}\ \frac{\varnothing \vdash e \rightsquigarrow \check{e}^{\Rightarrow}}{e \rightsquigarrow \overset{\square}{\Rightarrow}_{\checkmark}\check{e}^{\Rightarrow}}$$

Fig. 2. Marking (from scratch)

3.2 Marking

Marking in MALC is given by the synthetic and analytic marking judgments defined in Figure 2. Note that the synthetic marking judgment $\Gamma \vdash e \rightsquigarrow \check{e}^{\Rightarrow}$ does not output the synthesized type because it is in the type annotation on $\check{e}^{\Rightarrow}$. Also note that the analytic judgment $\Gamma \vdash \tau \Rightarrow e \rightsquigarrow {}^{\Rightarrow}\check{e}$ has the type on the left; we use this notation so information flows from left to right.

The MarkHole rule marks a bare expression hole, ?, as an expression hole synthesizing the unknown type. MarkConst similarly marks a constant synthesizing the base type.

The MarkVar rule marks a variable expression with a synthesized type found by looking up the variable in the context. Unlike in MLC, the context lookup judgment $x_m : \tau \in \Gamma$, defined in Figure 3, is a total function of x and Γ, returning m and τ. If x is an entry in Γ then $m = \checkmark$ because the x is not free, and the associated τ is returned; if not, then $m = \times$ and $\tau = ?$.

The MarkAsc rule marks a type ascription by analytically marking the expression body e against the ascribed type τ, resulting in ${}^{\Rightarrow}\check{e}$, and synthesizing τ as the type of the whole expression.

The MarkSynFun rule synthesizes a type for a function abstraction by first synthetically marking the body in the extended context. Note the extension of the context with a binder $x?$, which is

$$\boxed{x_m : \tau \in \Gamma}$$

$$\frac{}{x_{\times} : ? \in \emptyset} \qquad \frac{}{x_{\checkmark} : \tau \in \Gamma,\ x : \tau} \qquad \frac{x \neq x' \qquad x_m : \tau \in \Gamma}{x_m : \tau \in \Gamma,\ x' : \tau'}$$

$$\boxed{\tau \blacktriangleright^{\rightarrow}_{m} \tau \rightarrow \tau}$$

$$\frac{}{? \blacktriangleright^{\rightarrow}_{\checkmark} ? \rightarrow ?} \qquad \frac{}{\tau_1 \rightarrow \tau_2 \blacktriangleright^{\rightarrow}_{\checkmark} \tau_1 \rightarrow \tau_2} \qquad \frac{\tau \neq ? \qquad \mathsf{hd}(\tau) \neq \rightarrow}{\tau \blacktriangleright^{\rightarrow}_{\times} ? \rightarrow ?}$$

$$\boxed{\tau \sim_m \tau}$$

$$\frac{}{? \sim_{\checkmark} \tau} \qquad \frac{}{\tau \sim_{\checkmark} ?}$$

$$\frac{\tau_1 \sim_{m_1} \tau_3 \qquad \tau_2 \sim_{m_2} \tau_4 \qquad m_1 \sqcap m_2 = m}{\tau_1 \rightarrow \tau_2 \sim_m \tau_3 \rightarrow \tau_4} \qquad \frac{\tau_1 \neq ? \qquad \tau_2 \neq ? \qquad \mathsf{hd}(\tau_1) \neq \mathsf{hd}(\tau_2)}{\tau_1 \sim_{\times} \tau_2}$$

Fig. 3. Total Side Conditions

either ? or some x. In the former case, since a binding hole does not bind any variables, we define $\Gamma,\ ? : \tau = \Gamma$. In the latter case, we interpret $\Gamma,\ x : \tau$ as simply extending an association list. The marked result in MarkSynFun is a marked abstraction with both marks set to ✓, because these mark positions indicate errors related to the analyzed type of the abstraction, and this is the synthetic rule. For the same reason, the body of the abstraction is analyzed against □, that is, it is not analyzed against any type. Finally, the abstraction synthesizes a function type with its domain determined by the annotation and its codomain determined by the body's synthesized type.

MarkAp reflects the standard bidirectional typing rule for function applications, with the added step of the *matched arrow judgment* from gradual type theory, defined in Figure 3. The expression in function position is marked in synthetic mode, obtaining synthesized type τ. Then τ is matched against the arrow type to obtain $\tau_1 \rightarrow \tau_2$ and mark m. This matching judgment is a necessary step for two independent reasons: because the unknown type ? is convertible to $? \rightarrow\ ?$, and because side conditions in this version of the calculus are total functions that return marks indicating success. The unknown type and an arrow type match the arrow form successfully, returning mark ✓, while any other type fails to match, returning the error mark × and unknown types.

Now we proceed to the analytic marking rules. Most syntactic forms do not have special typing behavior when typed in analytic mode; they simply ignore the analyzed type, mark in synthetic mode, and compare the analyzed and synthesized types. Such forms are called *subsumable*, and include all forms in this minimal language except function abstractions. The analytic marking rule for these forms, MarkSubsume, first marks the expression in synthetic mode, then checks the consistency of the analyzed and synthesized types. The consistency judgment checks whether two types match, up to the replacement of some subterms with the unknown type. The rules are given in Figure 3. The rule for arrow types uses the expression $m_1 \sqcap m_2$, which is defined by the property that $m_1 \sqcap m_2 = \checkmark$ if and only if $m_1 = \checkmark$ and $m_2 = \checkmark$. The only output of the consistency judgment is the mark. In the case of MarkSubsume, this mark is placed on the marked analytic expression.

The MarkAnaFun rule specifies how abstractions are marked in analytic mode. The analyzed type is matched against the arrow type, with the resulting mark placed on the abstraction, because it is a type error to find a function abstraction where a non-function was expected. The resulting domain type is checked for consistency with the abstraction's annotation, resulting in the second mark placed on the abstraction, because the expected and found input types must match. Finally,

$$\begin{array}{llll}
c & \in & \mathsf{Child} & \mathsf{One} \mid \mathsf{Two} \\
\alpha & \in & \mathsf{SimpleAction} & \mathsf{InsertConst} \mid \mathsf{InsertVar}(x) \mid \mathsf{WrapFun} \mid \mathsf{WrapAp}(c) \mid \mathsf{WrapAsc} \\
& & & \mid\ \mathsf{Delete} \mid \mathsf{Unwrap}(c) \mid \mathsf{SetAnn}(\tau) \mid \mathsf{SetAsc}(\tau) \\
& & & \mid\ \mathsf{InsertBinder}(x?) \mid \mathsf{DeleteBinder} \\
\bar{s} & \in & \mathsf{List}[\mathsf{Sort}] & \cdot \mid s, \bar{s} \\
A & \in & \mathsf{LocalizedAction} & (\alpha,\ \bar{c})
\end{array}$$

Fig. 4. Actions

the body is marked in analytic mode against the codomain of the analyzed type. Note that the mark on the marked analytic expression form is always set to ✓ in this case, rendering it redundant when it appears before a function abstraction. This redundancy could be eliminated by removing the mark from the marked analytic expression form and giving every subsumable constructor expression form its own consistency mark, but this would be onerous to manage. We accept the redundancy for the sake of the convenient factorization of the subsumption rule.

This rule could be modified so that the analyzed type informs the type of the bound variable, for example by giving it the join of the annotated type and the analyzed type. This would provide a limited form of variable type inference in our bidirectional setting, without requiring unification. We forwent this for simplicity, as it would not interact interestingly with the rest of the theory.

A bare expression is marked as a program by marking it in synthetic mode in the empty context.

The marking operation from bare expressions to marked programs induces a notion of *well-markedness* on marked programs. Intuitively, a marked program is well-marked if it is the result of marking its underlying bare expression correctly. To formalize this, we utilize a *marking erasure* function, denoted $|\check{p}| = e$. Erasure recursively removes marks and synthesized and analyzed types, resulting in the underlying bare expression. Well-markedness is then defined as such:

Definition 3.1 (Well-Markedness). A marked program $\check{p}$ is *well-marked* if $|\check{p}| \rightsquigarrow \check{p}$.

Well-markedness is used to express the correctness of Incremental MALC in Section 5.

3.3 Edit Actions

Let us now consider how users edit terms in the language. This is formalized in an *action calculus*, consisting of a set of simple actions α, representing the kinds of edits the user can make, a set of localized actions A, consisting of simple actions paired with a location in the program, and an action performance judgment $e_1 \xrightarrow{A} e_2$, which performs a localized action in a bare expression.

Figure 4 defines the syntax of simple edit actions. They consist of actions to insert constructors at a leaf ($\mathsf{InsertVar}(x)$, $\mathsf{InsertConst}$), "wrap" actions to insert new constructors as parents of a subterm ($\mathsf{WrapFun}$, $\mathsf{WrapAp}(c)$, $\mathsf{WrapAsc}$), an action to delete a subterm (Delete), an "unwrap" action to delete the parent constructor and sibling subterms of a subterm ($\mathsf{Unwrap}(c)$), actions to change types ($\mathsf{SetAnn}(\tau)$, $\mathsf{SetAsc}(\tau)$), and actions to insert or delete binders ($\mathsf{InsertBinder}(x?)$, $\mathsf{DeleteBinder}$). (Un)wrapping a constructor with multiple children requires specifying which child position to (un)wrap unwrap around, hence the child argument to these actions, c. We do not model fine grained type edits, since these are independent of the incremental type maintenance behavior, but abstract them into wholesale type edit actions $\mathsf{SetAnn}(\tau)$ and $\mathsf{SetAsc}(\tau)$.

A localized action A is a simple action α paired with a path into the expression from the root, represented as a list of child elements. Not every such path identifies a valid subexpression.

The action performance judgment $e_1 \xrightarrow{A} e_2$ states that the performing the localized action A on bare expression e_1 results in e_2. The rules for this judgment are straightforward, and the formal definition can be found in the Agda mechanization. While the choice of action language is an input to the theory, we are able to validate our choice in one respect by proving the completeness of our set of actions; any program structure can be reached from any other by some sequence of actions:

Theorem 3.2 (Action Completeness). *For any bare expressions e_1 and e_2, there exists a sequence of localized actions $\overline{A}$ such that $e_1 \xrightarrow{\overline{A}} e_2$ (where $e_1 \xrightarrow{\overline{A}} e_2$ denotes iterated action performance).*

4 Incremental MALC By Example

Before formally defining Incremental MALC, let us develop some intuition by following a small example edit trace. In Incremental MALC, programs are represented as incremental expressions, which mirror the marked expressions of the previous section but attach to each type a dirty bit, either *dirty*, *, or *clean*, $^\bullet$, which mediates the propagation of type information updates. The beginning of each subsection below displays the edit in a syntax closer to what the user would see, omitting type information except for occurrences of $\times$. Note that only the names of *simple* edit actions are displayed, with the location of the edit implied by its effect. For reference, the arrows representing steps in this section are tagged with names of the corresponding inference rules in section 5.

Term (1) displays the initial state of the program, which begins as an empty expression hole synthesizing the unknown type (?) and not analyzed against a type ($\square$). Since there is no inconsistency between the analyzed and synthesized types, there is no consistency error on this expression. The types are furthermore annotated with $^\bullet$ symbols, indicating that no type information needs to be propagated through the program at this time.

$$\overset{\square^\bullet}{\Rightarrow}_{\checkmark} ? \overset{?^\bullet}{\Rightarrow} \tag{1}$$

4.1 Inserting a Variable

$$\boxed{? \xrightarrow{\mathsf{InsertVar}(x)} x_{\times}}$$

Suppose the user's first action is to insert an identifier x into the empty hole. This is modeled as the performance of the action $\mathsf{InsertVar}(x)$, localized at the hole. The immediate result is term (2).

$$\xrightarrow{\mathsf{InsertVar}(x)} \qquad \overset{\square^\star}{\Rightarrow}_{\checkmark} x_{\times} \overset{?^\star}{\Rightarrow} \tag{2}$$

The variable x is free at this location in the program, so it is given an error mark $\times$. Additionally, the analyzed and synthesized types are dirtied, meaning they have been added to the update propagation frontier. The analyzed type, though unchanged, is dirtied because the expression being analyzed is new. The synthesized type of the new variable is set to the variable's type. As it happens in this case, since the variable is free, it again synthesizes the unknown type.

At this point the editor, perhaps while waiting for the user's next action, would take two steps to propagate this new type information. The update propagation dynamics is nondeterministic, allowing either of the two dirtied types to step first. There is, however, a preferred prioritization order that minimizes redundant computation in the general case, as discussed in Section 6.3. In this running example, update propagation steps are taken in this order.

First the dirty analyzed type takes a step, bringing us to term (3).

$$\mapsto_{(\textsc{StepAna})} \qquad \overset{\square^\bullet}{\Rightarrow}_{\checkmark} x_{\times} \overset{?^\star}{\Rightarrow} \tag{3}$$

The analyzed type encounters a subsumable syntactic form, so it updates the consistency mark, which in this case remains $\checkmark$. The analyzed $\square^\star$ is cleaned since its immediate ramifications have been calculated; it is no longer on the frontier.

The synthesized type, being at the program root, simply exits the frontier, bringing us to term 4, which has no dirty types. Such a term is called *quiescent*.

$$\mapsto_{(\textsc{TopStep})} \qquad \overset{\square^{\bullet}}{\Rightarrow}_{\checkmark} x_{\times} \overset{?^{\bullet}}{\Rightarrow} \tag{4}$$

4.2 Wrapping a Function Application

$$\boxed{x_{\times} \xrightarrow{\text{WrapAp(One), InsertNum(1)}} x_{\times} \lhd 1}$$

Next, the user wraps the current expression in a function application form, then inserts a number literal into the argument position before update propagation. The result is shown in term (5). Note that base types are assumed for this example, and included in the mechanization.

$$\xrightarrow{\text{WrapAp(One) ... InsertNum(1)}} \qquad \overset{\square^{\star}}{\Rightarrow}_{\checkmark} \left(\overset{\square^{\star}}{\Rightarrow}_{\checkmark} x_{\times} \overset{?^{\star}}{\Rightarrow} \right) \lhd_{\checkmark} \left(\overset{\square^{\star}}{\Rightarrow}_{\checkmark} 1 \overset{\text{num}^{\star}}{\Rightarrow} \right) \overset{\square^{\star}}{\Rightarrow} \tag{5}$$

The leftmost two $\square^{\star}$ propagate similarly to the step from term (2) to term (3), resulting in term (6).

$$\mapsto_{(\textsc{StepAna})} \mapsto_{(\textsc{StepAna})} \qquad \overset{\square^{\bullet}}{\Rightarrow}_{\checkmark} \left(\overset{\square^{\bullet}}{\Rightarrow}_{\checkmark} x_{\times} \overset{?^{\star}}{\Rightarrow} \right) \lhd_{\checkmark} \left(\overset{\square^{\star}}{\Rightarrow}_{\checkmark} 1 \overset{\text{num}^{\star}}{\Rightarrow} \right) \overset{\square^{\star}}{\Rightarrow} \tag{6}$$

Next the synthesized type from the function position of the application propagates outwards. The unknown type is matched against the function type as $? \to \; ?$. The argument expression is analyzed against the domain type ?, and the entire application form synthesizes the codomain type ?. This new analytic and synthetic type data is dirtied. The application's error mark remains $\checkmark$, because matched function type succeeded. The result is term (7).

$$\mapsto_{(\textsc{StepAp})} \qquad \overset{\square^{\bullet}}{\Rightarrow}_{\checkmark} \left(\overset{\square^{\bullet}}{\Rightarrow}_{\checkmark} x_{\times} \overset{?^{\bullet}}{\Rightarrow} \right) \lhd_{\checkmark} \left(\overset{?^{\star}}{\Rightarrow}_{\checkmark} 1 \overset{\text{num}^{\star}}{\Rightarrow} \right) \overset{?^{\star}}{\Rightarrow} \tag{7}$$

The newly analyzed $?^{\star}$ against the subsumable number literal in the argument position is then propagated, setting the consistency mark to $\checkmark$ again because the analyzed ? is consistent with the synthesized num. The synthesized $?^{\star}$ exits the frontier, having reached the root of the program. These two steps result in the quiescent program shown in term (8).

$$\mapsto_{(\textsc{StepAna})} \mapsto_{(\textsc{StepSyn})} \mapsto_{(\textsc{TopStep})} \qquad \overset{\square^{\bullet}}{\Rightarrow}_{\checkmark} \left(\overset{\square^{\bullet}}{\Rightarrow}_{\checkmark} x_{\times} \overset{?^{\bullet}}{\Rightarrow} \right) \lhd_{\checkmark} \left(\overset{?^{\bullet}}{\Rightarrow}_{\checkmark} 1 \overset{\text{num}^{\bullet}}{\Rightarrow} \right) \overset{?^{\bullet}}{\Rightarrow} \tag{8}$$

4.3 Wrapping a Function Abstraction

$$\boxed{x_{\times} \lhd 1 \xrightarrow{\text{WrapFun, SetAnn(bool}\to\text{num), InsertBinder}(x)} \lambda(x \,:\, (\text{bool} \to \text{num})).\ (x \lhd {}_{\times}1)}$$

Now the user wraps the program in a function abstraction, resulting in term (9). The binding position of the abstraction as well as its type annotation are initially empty holes, and a few types have been added to the update propagation frontier.

$$\xrightarrow{\text{WrapFun}} \qquad \overset{\square^{\star}}{\Rightarrow}_{\checkmark} \lambda(? \,:\, ?^{\bullet})_{\checkmark,\checkmark}. \left(\overset{\square^{\star}}{\Rightarrow}_{\checkmark} \left(\overset{\square^{\bullet}}{\Rightarrow}_{\checkmark} x_{\times} \overset{?^{\bullet}}{\Rightarrow} \right) \lhd_{\checkmark} \left(\overset{?^{\bullet}}{\Rightarrow}_{\checkmark} 1 \overset{\text{num}^{\bullet}}{\Rightarrow} \right) \overset{?^{\star}}{\Rightarrow} \right) \overset{\square^{\star}}{\Rightarrow} \tag{9}$$

Before update propagation, suppose the annotation is edited to bool → num. This new type in the surface syntax is placed in the update propagation frontier.

$$\xrightarrow{\text{SetAnn(bool}\to\text{num)}} \quad \overset{\square^{\star}}{\Rightarrow}_{\checkmark} \lambda(? \,:\, (\text{bool} \to \text{num})^{\star})_{\checkmark,\checkmark}. \left(\overset{\square^{\star}}{\Rightarrow}_{\checkmark} \left(\overset{\square^{\bullet}}{\Rightarrow}_{\checkmark} x_{\times} \overset{?^{\bullet}}{\Rightarrow} \right) \lhd_{\checkmark} \left(\overset{?^{\bullet}}{\Rightarrow}_{\checkmark} 1 \overset{\text{num}^{\bullet}}{\Rightarrow} \right) \overset{?^{\star}}{\Rightarrow} \right) \overset{\square^{\star}}{\Rightarrow} \tag{10}$$

Still before any update propagation steps, let the user insert the variable x in the binding position of the abstraction. This demonstrates an important property of the calculus: bindings, being nonlocal connections in the analysis of the program, are not updated via update propagation steps. They are updated atomically as part of the edit action that affects them. In this case, as soon as a binding for x is inserted around the occurrence of x, the occurrence's error mark is set to $\checkmark$, since it is no longer free, and it synthesizes the type dictated by the binding annotation.

$$\xrightarrow{\mathsf{InsertBinder}(x)} \overset{\square^\star}{\Rightarrow}_{\checkmark}\lambda(x : (\mathsf{bool}\to\mathsf{num})^\star)_{\checkmark,\checkmark}.\left(\overset{\square^\star}{\Rightarrow}_{\checkmark}\left(\overset{\square^\bullet}{\Rightarrow}_{\checkmark} x_{\checkmark}\overset{(\mathsf{bool}\to\mathsf{num})^\star}{\Rightarrow}\right)\lhd_{\checkmark}\left(\overset{?^\bullet}{\Rightarrow}_{\checkmark}1\overset{\mathsf{num}^\bullet}{\Rightarrow}\right)\overset{?^\star}{\Rightarrow}\right)\overset{\square^\star}{\Rightarrow} \tag{11}$$

It is a challenge to efficiently maintain type information in response to binding changes such as the one above; the most obvious strategies require traversing the body of a binding construct or traversing the program's spine above a variable occurrence, but these operations may incur costs linear in the size of the program. It is this task to which we apply the order maintenance data structure, as described in detail in Section 6. Formally, however, binding calculations such as updating the bound x above are assumed to occur atomically during action performance.

Now let us consider the propagation of updates in the current program state. The first two steps propagate the analyzed type and the annotation on the abstraction, resulting in a new analyzed type for the body and a new synthesized type for the abstraction. Function abstractions, being non-subsumable, use the expected type to find the expected type of the body, and only synthesize a type if not analyzed against a type. The result of these first two steps is shown in term (12).

$$\mapsto(\textsc{StepAnnFun})\mapsto(\textsc{StepAnaFun})$$

$$\overset{\square^\bullet}{\Rightarrow}_{\checkmark}\lambda(x : (\mathsf{bool}\to\mathsf{num})^\bullet)_{\checkmark,\checkmark}.\left(\overset{\square^\star}{\Rightarrow}_{\checkmark}\left(\overset{\square^\bullet}{\Rightarrow}_{\checkmark} x_{\checkmark}\overset{(\mathsf{bool}\to\mathsf{num})^\star}{\Rightarrow}\right)\lhd_{\checkmark}\left(\overset{?^\bullet}{\Rightarrow}_{\checkmark}1\overset{\mathsf{num}^\bullet}{\Rightarrow}\right)\overset{?^\star}{\Rightarrow}\right)\overset{((\mathsf{bool}\to\mathsf{num})\to ?)^\star}{\Rightarrow} \tag{12}$$

Next the analyzed type on the variable is checked against the synthesized type, and then the synthesized type propagates through the function application. The result of these two steps is shown in term (13).

$$\mapsto(\textsc{StepAna})\mapsto(\textsc{StepAp})$$

$$\overset{\square^\bullet}{\Rightarrow}_{\checkmark}\lambda(x : (\mathsf{bool}\to\mathsf{num})^\bullet)_{\checkmark,\checkmark}.\left(\overset{\square^\bullet}{\Rightarrow}_{\checkmark}\left(\overset{\square^\bullet}{\Rightarrow}_{\checkmark} x_{\checkmark}\overset{(\mathsf{bool}\to\mathsf{num})^\bullet}{\Rightarrow}\right)\lhd_{\checkmark}\left(\overset{\mathsf{bool}^\star}{\Rightarrow}_{\checkmark}1\overset{\mathsf{num}^\bullet}{\Rightarrow}\right)\overset{\mathsf{num}^\star}{\Rightarrow}\right)\overset{((\mathsf{bool}\to\mathsf{num})\to ?)^\star}{\Rightarrow} \tag{13}$$

The analyzed type on the argument is compared with the synthesized type, and the synthesized type of the body propagates to a new synthesized type for the abstraction. The result is term (14).

$$\mapsto(\textsc{StepAna})\mapsto(\textsc{StepSynFun})$$

$$\overset{\square^\bullet}{\Rightarrow}_{\checkmark}\lambda(x : (\mathsf{bool}\to\mathsf{num})^\bullet)_{\checkmark,\checkmark}.\left(\overset{\square^\bullet}{\Rightarrow}_{\checkmark}\left(\overset{\square^\bullet}{\Rightarrow}_{\checkmark} x_{\checkmark}\overset{(\mathsf{bool}\to\mathsf{num})^\bullet}{\Rightarrow}\right)\lhd_{\checkmark}\left(\overset{\mathsf{bool}^\bullet}{\Rightarrow}_{\times}1\overset{\mathsf{num}^\bullet}{\Rightarrow}\right)\overset{\mathsf{num}^\bullet}{\Rightarrow}\right)\overset{((\mathsf{bool}\to\mathsf{num})\to\mathsf{num})^\star}{\Rightarrow} \tag{14}$$

Finally, the synthesized type of the whole program exits the update propagation frontier, having reached the root. The result is the quiescent program shown in term (15).

$$\mapsto(\textsc{TopStep})$$

$$\overset{\square^\bullet}{\Rightarrow}_{\checkmark}\lambda(x : (\mathsf{bool}\to\mathsf{num})^\bullet)_{\checkmark,\checkmark}.\left(\overset{\square^\bullet}{\Rightarrow}_{\checkmark}\left(\overset{\square^\bullet}{\Rightarrow}_{\checkmark} x_{\checkmark}\overset{(\mathsf{bool}\to\mathsf{num})^\bullet}{\Rightarrow}\right)\lhd_{\checkmark}\left(\overset{\mathsf{bool}^\bullet}{\Rightarrow}_{\times}1\overset{\mathsf{num}^\bullet}{\Rightarrow}\right)\overset{\mathsf{num}^\bullet}{\Rightarrow}\right)\overset{((\mathsf{bool}\to\mathsf{num})\to\mathsf{num})^\bullet}{\Rightarrow} \tag{15}$$

This example, though small, illustrates the fundamental ideas behind Incremental MALC. Edit actions locally update annotations and place them in the propagation frontier, and a small-step dynamics propagates these changes through the rest of the program in a series of local steps. In this syntax these updates propagate from left to right, an idea which has been formalized to prove termination for the dynamics. Once a program is quiescent and no more steps are possible, its marks and type annotations will be correct with respect to the non-incremental MALC.

$$
\begin{array}{lcll}
\circ & \in & \mathsf{DirtyBit} & \star \mid \bullet \\
e^{\Rightarrow} & \in & \mathsf{SynExp} & \dot{e} \overset{\sigma^{\circ}}{\Rightarrow} \\
\dot{e} & \in & \mathsf{ConExp} & ? \mid \mathsf{c} \mid x_m \mid \lambda(x? : \tau^{\circ})_{m,m}. {}^{\Rightarrow}e \mid {}^{\Rightarrow}e \lhd_m {}^{\Rightarrow}e \mid {}^{\Rightarrow}e : \tau^{\circ} \\
{}^{\Rightarrow}e & \in & \mathsf{AnaExp} & \overset{\sigma^{\circ}}{\Rightarrow}_m e^{\Rightarrow} \\
p & \in & \mathsf{Program} \subseteq \mathsf{AnaExp} & \overset{\Box^{\circ}}{\Rightarrow}_{\checkmark} e^{\Rightarrow}
\end{array}
$$

Fig. 5. Incremental Syntax

$$\overline{A} \qquad \cdot \mid A, \overline{A}$$

$$
\frac{p_1 \xrightarrow{A} p_2 \qquad p_2 \overset{\overline{A}}{\longmapsto} p_3}{p_1 \overset{A,\overline{A}}{\longmapsto} p_3}
\qquad
\frac{p_1 \mapsto p_2 \qquad p_2 \overset{\overline{A}}{\longmapsto} p_3}{p_1 \overset{\overline{A}}{\longmapsto} p_3}
\qquad
\frac{\neg \exists p'.\ p \mapsto p'}{p \overset{\cdot}{\longmapsto} p}
$$

Fig. 6. Interleaved action and update propagation

5 Incremental MALC

We now formally specify Incremental MALC, introduced by example in the previous section. Section 5.1 and Section 5.2 describe the syntax, central judgments, and main metatheoretic properties of the incremental theory. Section 5.3 and Section 5.4 explain the detailed logic of action performance and update propagation. Section 5.5 describes the general form of the approach, and includes an extension to System F style polymorphism. Section 5.6 discusses the Agda mechanization.

5.1 Incremental Program Syntax

Incremental MALC operates on *incremental programs*, the syntax for which is defined in Figure 5.

Incremental programs are the same as the marked programs of Section 3, except that the types within, both those appearing in the surface syntax and stored analyzed or synthesized types, are additionally annotated with a dirty bit, either *dirty*, *, or *clean*, $^{\bullet}$. Well-markedness is defined on incremental programs by coercing them to marked programs by erasing the dirty bits.

Types marked as dirty with * are members of the update propagation frontier, and represent a location where an update step is possible. Usually $\sigma^{\star}$ indicates that σ has recently been set to a possibly new value, and the immediate ramifications of this value have not been computed yet. However, this is not always the case. For example, an analyzed type is annotated with * when the expression beneath it changes, rather than when the type itself changes.

5.2 Central Judgments and Properties

These are the two central judgment forms of the calculus. The action performance judgment, written $p \xrightarrow{A} p'$, applies a localized edit action A to the program p, resulting in new program p'. The update propagation judgment, written $p \mapsto p'$, describes the small step propagation dynamics that cause updates to static information throughout the program.

The judgment $p \overset{\overline{A}}{\longmapsto} p'$, defined in Figure 6, combines action performance and update propagation to model the top-level behavior of the system. In the editor, propagations are run automatically and concurrently with action performance until none are possible. The combined judgment applies all actions in sequence, interleaved nondeterministically with arbitrary update propagation steps, until no actions remain and no steps are possible.

The primary correctness properties of this system are the following. The proofs are discussed further in the appendix, and the full proofs are mechanized in Agda, as described in Section 5.6.

Validity ensures that the incremental analysis produces correct results with respect to MALC. Well-formedness is an invariant that is preserved by actions and update steps. It is formally specified in the appendix. Informally, a program is well-formed if its type information is locally correct according to the MALC, except possibly on the update propagation frontier.

Theorem 5.1 (Validity). *If program p is well-formed and $p \overset{\overline{A}}{\longmapsto} p'$, then p' is well-marked.*

Convergence guarantees that update propagation steps can occur at any time during editing, and in any order, including interleaved with edits, and the same annotated program will result.

Theorem 5.2 (Convergence). *If program p is well-formed, $p \overset{\overline{A}}{\longmapsto} p_1$, and $p \overset{\overline{A}}{\longmapsto} p_2$, then $p_1 = p_2$.*

Termination guarantees that update propagation cannot continue forever, which guarantees the eventual correctness of the analysis provided a finite number of edit actions have been performed.

Theorem 5.3 (Termination). *There is no infinite sequence $\{p_n\}_{n=0}^{\infty}$ such that $\forall n.\ p_n \mapsto p_{n+1}$.*

5.3 Action Performance

The action performance judgment for analytic expressions, $\Gamma \vdash {}^{\Rightarrow}e_1 \xrightarrow{\alpha} {}^{\Rightarrow}e_2$, means that performing the simple action α directly on the analytic expression ${}^{\Rightarrow}e_1$, appearing in the typing context Γ, results in the new analytic expression ${}^{\Rightarrow}e_2$. It is defined by only one rule, which marks the analyzed type as new and applies the action to the synthetic expression within:

$$\textsc{ActAna}\quad \frac{\Gamma \vdash e_1^{\Rightarrow} \xrightarrow{\alpha} e_2^{\Rightarrow}}{\Gamma \vdash \overset{\sigma^{\circ}}{\Rightarrow}_m e_1^{\Rightarrow} \xrightarrow{\alpha} \overset{\sigma^{\star}}{\Rightarrow}_m e_2^{\Rightarrow}}$$

The corresponding judgment for synthetic expressions, $\Gamma \vdash e_1^{\Rightarrow} \xrightarrow{\alpha} e_2^{\Rightarrow}$, is where the core logic of action performance is specified. Each rule is designed to satisfy a few criteria. First, each action must change the structure of the program correctly (e.g. deletion should actually delete the subterm). This is formalized in the following lemma, which states that erasure maps action performance annotated expressions to action performance on bare expressions.

Lemma 5.4 (Action Erasure). *If $p \xrightarrow{A} p'$, then $|p| \xrightarrow{A} |p'|$.*

Secondly, each action must preserve the well-formedness invariant, the full definition of which can be found in the appendix. This allows us to prove the following essential lemma:

Lemma 5.5 (Action Preservation). *If p is well-formed and $p \xrightarrow{A} p'$, then p' is well-formed.*

An additional criterion that is not essential for the formal properties of the system but is important for the practical instantiation of the system is that actions may add but never remove types from the update propagation frontier. Let us consider each rule in turn.

$$\textsc{ActInsertConst}\quad \frac{}{\Gamma \vdash ?\overset{\sigma^{\circ}}{\Rightarrow} \xrightarrow{\mathsf{InsertConst}} \mathsf{c}\overset{\mathsf{b}^{\star}}{\Rightarrow}} \qquad \textsc{ActInsertVar}\quad \frac{x_m : \tau^{\circ} \in \Gamma}{\Gamma \vdash ?\overset{\sigma^{\circ}}{\Rightarrow} \xrightarrow{\mathsf{InsertVar}(x)} x_m\overset{\tau^{\star}}{\Rightarrow}}$$

The rule ActInsertConst simply inserts a constant expression form into a hole, setting the synthesized type to the base type. The synthesized type is added to the update propagation frontier.

ActInsertVar similarly inserts a variable. Formally, the context is queried to find the variable's mark and synthesized type, just as in the non-incremental MALC.

$$\frac{\textsc{ActWrapFun}}{\Gamma \vdash \dot{e} \overset{\sigma^{\circ}}{\Rightarrow} \xrightarrow{\text{WrapFun}} \lambda(? : ?^{\bullet})_{\checkmark,\checkmark}.\left(\overset{\square^{\star}}{\Rightarrow}_{\checkmark}\dot{e} \overset{\sigma^{\bullet}}{\Rightarrow}\right)\overset{\square^{\star}}{\Rightarrow}}$$

The rule ActWrapFun wraps a subexpression in a lambda abstraction, with the binder and annotation initialized as holes. The synthesized type of the form is added to the update propagation frontier even though update propagation flowing from the other $\square^{\star}$ will eventually overwrite it. If it were not added to the frontier, the program would settle into the same state, and the correctness properties of the system would not be endangered. It is added to satisfy the well-formedness invariant, which is defined in terms of local consistency between adjacent data in the syntax tree. In other words, it is added not to guarantee correctness but to support the proof of correctness.

$$\frac{\textsc{ActWrapApOne}}{\Gamma \vdash \dot{e} \overset{\sigma^{\circ}}{\Rightarrow} \xrightarrow{\text{WrapAp(One)}} \left(\overset{\square^{\star}}{\Rightarrow}_{\checkmark}\dot{e} \overset{\sigma^{\star}}{\Rightarrow}\right) \lhd_{\checkmark} \left(\overset{\square^{\bullet}}{\Rightarrow}_{\checkmark}? \overset{?^{\bullet}}{\Rightarrow}\right)\overset{\square^{\star}}{\Rightarrow}}$$

The rule ActWrapApOne wraps a subexpression in the function position of a function application form. The argument position is filled with a hole. Following the ordinary marking rule, the function position is not analyzed against a type.

$$\frac{\textsc{ActWrapApTwo}}{\Gamma \vdash \dot{e} \overset{\sigma^{\circ}}{\Rightarrow} \xrightarrow{\text{WrapAp(Two)}} \left(\overset{\square^{\bullet}}{\Rightarrow}_{\checkmark}? \overset{?^{\bullet}}{\Rightarrow}\right) \lhd_{\checkmark} \left(\overset{?^{\star}}{\Rightarrow}_{\checkmark}\dot{e} \overset{\sigma^{\bullet}}{\Rightarrow}\right)\overset{?^{\star}}{\Rightarrow}}$$

ActWrapApTwo is similar, wrapping the subexpression in the argument position instead. This rule "folds in" a step of update propagation by placing the argument's analyzed type and the application's synthesized type on the update propagation frontier, rather than just the function position's synthesized type.

$$\frac{\textsc{ActWrapAsc}}{\Gamma \vdash \dot{e} \overset{\sigma^{\circ}}{\Rightarrow} \xrightarrow{\text{WrapAsc}} \left(\overset{?^{\star}}{\Rightarrow}_{\checkmark}\dot{e} \overset{\sigma^{\circ}}{\Rightarrow}\right) : ?^{\bullet}\overset{?^{\star}}{\Rightarrow}} \qquad \frac{\textsc{ActDelete}}{\Gamma \vdash e^{\Rightarrow} \xrightarrow{\text{Delete}} ? \overset{?^{\star}}{\Rightarrow}}$$

ActWrapAsc wraps the subexpression in a type hole ascription, and also folds in the propagation of this new ascription to the neighboring analyzed and synthesized positions. ActDelete simply replaces an expression with an expression hole synthesizing the unknown type. Note that each occurrence of $\circ$ in these rules is a metavariable for a dirty bit value. To reduce visual clutter, we assume that these metavariables implicitly share the subscript of the type they accompany.

ActUnwrapFun

$$\frac{x?_{m_4} : \tau_2^{\circ} \in \Gamma \qquad [\![x?_{m_4}\overset{\tau_2}{\Rightarrow}/x?]\!]e^{\Rightarrow} = \dot{e}\overset{\sigma_3^{\circ}}{\Rightarrow}}{\Gamma \vdash \lambda(x? : \tau_1^{\circ})_{m_1,m_2}.\left(\overset{\sigma_1^{\circ}}{\Rightarrow}_{m_3}e^{\Rightarrow}\right)\overset{\sigma_2^{\circ}}{\Rightarrow} \xrightarrow{\text{Unwrap(One)}} \dot{e}\overset{\sigma_3^{\star}}{\Rightarrow}}$$

ActUnwrapFun unwraps a function abstraction from around its body. By deleting a binding location, this action has the potential to "unshadow" an outer binder for the same variable name, causing the occurrences of the variable that used to be bound to the deleted binder to be rebound to the outer binder. The first premise looks in the context for the type of the binder variable in the outer scope. The second premise uses a new judgment, the "variable update" judgment, defined in the appendix. Its notation evokes substitution because it sets the consistency mark and synthesized type of all free occurrences of $x?$ in $e^{\Rightarrow}$. The resulting synthetic expression, with the type marked as new, is the result of the action. Note the first premise is a variant of the context lookup judgment that accepts a binding argument, rather than just a variable. If $x? = \; ?$, this version of context lookup returns an arbitrary m and τ, and the variable update operation acts as the identity.

ActUnwrapApOne

$$\frac{}{\Gamma \vdash \left(\overset{\sigma_1^\circ}{\Rightarrow}_{m_1} \dot{e} \overset{\sigma_2^\circ}{\Rightarrow}\right) \lhd_{m_2} \overset{\Rightarrow}{e} \overset{\sigma_3^\circ}{\Rightarrow} \xrightarrow{\text{Unwrap(One)}} \dot{e} \overset{\sigma_2^\star}{\Rightarrow}}$$

ActUnwrapApTwo

$$\frac{}{\Gamma \vdash \overset{\Rightarrow}{e} \lhd_{m_1} \left(\overset{\sigma_1^\circ}{\Rightarrow}_{m_2} \dot{e} \overset{\sigma_2^\circ}{\Rightarrow}\right) \overset{\sigma_3^\circ}{\Rightarrow} \xrightarrow{\text{Unwrap(Two)}} \dot{e} \overset{\sigma_2^\star}{\Rightarrow}}$$

ActUnwrapAsc

$$\frac{}{\Gamma \vdash \left(\overset{\sigma_1^\circ}{\Rightarrow}_{m} \dot{e} \overset{\sigma_2^\circ}{\Rightarrow}\right) : \tau^\circ \overset{\sigma_3^\circ}{\Rightarrow} \xrightarrow{\text{Unwrap(One)}} \dot{e} \overset{\sigma_2^\star}{\Rightarrow}}$$

Rules ActUnwrapApOne and ActUnwrapApTwo unwrap an application form from around its left or right child, deleting the other child expression. The synthesized type of expression is added to the update propagation frontier. ActUnwrapAsc is analogous, deleting the type ascription.

ActSetAnn

$$\frac{}{\Gamma \vdash \lambda(x? : \tau_1^\circ)_{m_1,m_2}.\overset{\Rightarrow}{e}\overset{\sigma^\circ}{\Rightarrow} \xrightarrow{\text{SetAnn}(\tau_2)} \lambda(x? : \tau_2^\star)_{m_1,m_2}.\overset{\Rightarrow}{e}\overset{\sigma^\circ}{\Rightarrow}}$$

ActSetAsc

$$\frac{}{\Gamma \vdash \overset{\Rightarrow}{e} : \tau_1^\circ \overset{\sigma^\circ}{\Rightarrow} \xrightarrow{\text{SetAsc}(\tau_2)} \overset{\Rightarrow}{e} : \tau_2^\star \overset{\sigma^\circ}{\Rightarrow}}$$

ActSetAnn and ActSetAsc update types in the surface syntax of the program, either in function annotations or type ascriptions. Each merely sets the type to the action's argument and dirties it.

ActInsertBinder

$$\frac{[\![x_{\checkmark} \overset{\tau}{\Rightarrow} / x]\!] e^{\Rightarrow} = \dot{e} \overset{\sigma_3^\circ}{\Rightarrow}}{\Gamma \vdash \lambda(? : \tau^\circ)_{m_1,m_2}.\left(\overset{\sigma_1^\circ}{\Rightarrow}_{m_3} e^{\Rightarrow}\right) \overset{\sigma_2^\circ}{\Rightarrow} \xrightarrow{\text{InsertBinder}(x)} \lambda(x : \tau^\circ)_{m_1,m_2}.\left(\overset{\sigma_1^\circ}{\Rightarrow}_{m_3} \dot{e} \overset{\sigma_3^\star}{\Rightarrow}\right) \overset{\sigma_2^\circ}{\Rightarrow}}$$

ActInsertBinder inserts a variable name into the binder hole of a function abstraction. This captures all free occurrences of the variable in the body, which now must synthesize the annotated type and be marked as bound, as implemented in the variable update premise.

ActDeleteBinder

$$\frac{x?_{m_4} : \tau_2^\circ \in \Gamma \qquad [\![x?_{m_4} \overset{\tau_2}{\Rightarrow} / x?]\!] e^{\Rightarrow} = \dot{e} \overset{\sigma_3^\circ}{\Rightarrow}}{\Gamma \vdash \lambda(x? : \tau_1^\circ)_{m_1,m_2}.\left(\overset{\sigma_1^\circ}{\Rightarrow}_{m_3} e^{\Rightarrow}\right) \overset{\sigma_2^\circ}{\Rightarrow} \xrightarrow{\text{DeleteBinder}} \lambda(? : \tau_1^\circ)_{m_1,m_2}.\left(\overset{\sigma_1^\circ}{\Rightarrow}_{m_3} \dot{e} \overset{\sigma_3^\star}{\Rightarrow}\right) \overset{\sigma_2^\circ}{\Rightarrow}}$$

ActDeleteBinder deletes the current binder of a function abstraction. This rule is similar to ActUnwrapFun, with the same premises. The only difference is that it does not remove the abstraction form from the body, it just sets the binder to a hole. The reasoning is the same as for ActUnwrapFun, as deleting the binding variable results in a potential "unshadowing" of the variables in the body.

5.4 Update Propagation

Now we turn to update propagation steps. Like the action performance rules, each step is designed to satisfy certain criteria. First, steps cannot change the structure of the program. They are not edits, but automated calculations that trigger while the user is editing. This is formalized in the following lemma:

Lemma 5.6 (Update Step Erasure). *If* $p \mapsto p'$*, then* $|p| = |p'|$.

Second, like actions, steps must maintain the well-formedness invariant:

Lemma 5.7 (Update Step Preservation). *If* p *is well-formed and* $p \mapsto p'$*, then* p' *is well-formed.*

Third, to support the implementation, each step removes exactly one type from the update propagation frontier, but can add any number. Fourth, the steps must make progress towards termination, so that no infinite sequence of steps is possible. This manifests as each step progressing the frontier of new types along the "bidirectional information flow," which corresponds to the left-to-right order in the syntax.

At the top level, a program can either take an ordinary step as an analytic expression, or a special step only possible at the root. This TopStep rule allows a new type synthesized by the whole program to exit the update propagation frontier, having reached the root of the program.

InsideStep

$$\frac{\overset{\square_1^\circ}{\Rightarrow}_{\checkmark} e_1^{\Rightarrow} \mathrel{\|\!\!\mapsto} \overset{\square_2^\circ}{\Rightarrow}_{\checkmark} e_2^{\Rightarrow}}{\overset{\square_1^\circ}{\Rightarrow}_{\checkmark} e_1^{\Rightarrow} \mapsto \overset{\square_2^\circ}{\Rightarrow}_{\checkmark} e_2^{\Rightarrow}}$$

TopStep

$$\frac{}{\overset{\square^\circ}{\Rightarrow}_{\checkmark} \dot{e} \overset{\sigma^\star}{\Rightarrow} \mapsto \overset{\square^\circ}{\Rightarrow}_{\checkmark} \dot{e} \overset{\sigma^\bullet}{\Rightarrow}}$$

Now we turn our attention to steps for analytic and synthetic expressions, which are written with $\|\!\!\mapsto$ to distinguish them from program steps. These judgments form a kind of "mutual congruence," meaning that an outer expression may step by stepping within a subexpression, regardless of whether the outer expression is analytic or synthetic and whether the subexpression is analytic or synthetic. Let us consider the rules for stepping directly.

StepSyn

$$\frac{\tau \sim_{m_2} \sigma}{\overset{\tau^\bullet}{\Rightarrow}_{m_1} \dot{e} \overset{\sigma^\star}{\Rightarrow} \mathrel{\|\!\!\mapsto} \overset{\tau^\bullet}{\Rightarrow}_{m_2} \dot{e} \overset{\sigma^\bullet}{\Rightarrow}}$$

StepAna

$$\frac{\dot{e} \text{ subsumable} \qquad \sigma_1 \sim_{m_2} \sigma_2}{\overset{\sigma_1^\star}{\Rightarrow}_{m_1} \dot{e} \overset{\sigma_2^\circ}{\Rightarrow} \mathrel{\|\!\!\mapsto} \overset{\sigma_1^\bullet}{\Rightarrow}_{m_2} \dot{e} \overset{\sigma_2^\circ}{\Rightarrow}}$$

The StepSyn rule handles a newly synthesized type under an analyzed type that is not $\square$. When the term is actually being analyzed against a type, the synthesized type is only used to calculate the consistency mark, hence consistency is reevaluated and nothing is dirtied. The StepAna rule handles the alternative case in which consistency is recomputed, that being when the analyzed type is new. If the root of the expression $\dot{e}$ is not subsumable, then consistency should not be checked and this rule is not applicable. Otherwise, the analyzed and synthesized data are checked for consistency as in the previous rule.

StepAnaFun

$$\frac{\sigma_1 \blacktriangleright_{\overrightarrow{m_5}} \sigma_5 \to \sigma_6 \qquad \sigma_5 \sim_{m_6} \tau \qquad \mathsf{FunSyn}(\sigma_1, \tau, \sigma_3) \;=\; \sigma_7}{\begin{array}{c} \overset{\sigma_1^\star}{\Rightarrow}_{m_1} \lambda(x? \,:\, \tau^\circ)_{m_2,m_3}. \left(\overset{\sigma_2^\circ}{\Rightarrow}_{m_4} \dot{e} \overset{\sigma_3^\circ}{\Rightarrow} \right) \overset{\sigma_4^\circ}{\Rightarrow} \\ \mathrel{\|\!\!\mapsto} \overset{\sigma_1^\bullet}{\Rightarrow}_{\checkmark} \lambda(x? \,:\, \tau^\circ)_{m_5,m_6}. \left(\overset{\sigma_6^\star}{\Rightarrow}_{m_4} \dot{e} \overset{\sigma_3^\circ}{\Rightarrow} \right) \overset{\sigma_7^\star}{\Rightarrow} \end{array}}$$

StepAnaFun propagates a new analyzed type on a function abstraction. The first two premises correspond to those in the non-incremental MALC. The third premise uses a new function, FunSyn. This determines what type the abstraction should synthesize in terms of the analyzed type, the annotated type, and the type synthesized by the body. It is defined by the following equations:

$$\begin{array}{lcl} \mathsf{FunSyn}(\tau_1, \tau_2, \sigma) & = & \square \\ \mathsf{FunSyn}(\square, \tau_2, \square) & = & \square \\ \mathsf{FunSyn}(\square, \tau_1, \tau_2) & = & \tau_1 \to \tau_2 \end{array}$$

If the abstraction is analyzed against some type, then it does not synthesize a type. If it is not analyzed against a type, then it synthesizes the appropriate function type between the annotated type and the body's type (unless the body does not synthesize a type). By including all three inputs, this function provides the correct behavior for abstractions in both analytic and synthetic mode.

StepAnnFun

$$\frac{[\![x?\,{}_{\checkmark}\overset{\tau}{\Rightarrow}/x?]\!]e_1^{\Rightarrow} \;=\; e_2^{\Rightarrow}}{\begin{array}{l} \overset{\sigma_1^\circ}{\Rightarrow}_{m_1}\lambda(x? \,:\, \tau^\star)_{m_2,m_3}.\left(\overset{\sigma_2^\circ}{\Rightarrow}_{m_4}e_1^{\Rightarrow}\right)\overset{\sigma_3^\circ}{\Rightarrow} \\ \Vdash\!\!\!\rightarrow \overset{\sigma_1^\star}{\Rightarrow}_{m_1}\lambda(x? \,:\, \tau^\bullet)_{m_2,m_3}.\left(\overset{\sigma_2^\circ}{\Rightarrow}_{m_4}e_2^{\Rightarrow}\right)\overset{\sigma_3^\circ}{\Rightarrow} \end{array}}$$

StepSynFun

$$\frac{\mathsf{FunSyn}(\sigma_1, \tau, \sigma_2) \;=\; \sigma_4}{\begin{array}{l} \overset{\sigma_1^\circ}{\Rightarrow}_{m_1}\lambda(x? \,:\, \tau^\circ)_{m_2,m_3}.\left(\overset{\square^\circ}{\Rightarrow}_{m_4}\dot{e}\overset{\sigma_2^\star}{\Rightarrow}\right)\overset{\sigma_3^\circ}{\Rightarrow} \\ \Vdash\!\!\!\rightarrow \overset{\sigma_1^\circ}{\Rightarrow}_{m_1}\lambda(x? \,:\, \tau^\circ)_{m_2,m_3}.\left(\overset{\square^\circ}{\Rightarrow}_{\checkmark}\dot{e}\overset{\sigma_2^\bullet}{\Rightarrow}\right)\overset{\sigma_4^\star}{\Rightarrow} \end{array}}$$

StepAnnFun propagates a change in the annotated type of an abstraction. The new type causes a corresponding update in the synthesized types of the abstraction's bound variables, which is implemented by the variable update in the premise. The new annotation also affects the synthesized type and second mark of the abstraction. For convenience, this is handled by marking the analyzed type as new, allowing StepAnaFun to trigger later and implement this logic, rather than repeating the premises in both rules. StepSynFun propagates a newly synthesized type from the body of a function abstraction upwards by recomputing FunSyn on the new inputs.

StepAp

$$\frac{\sigma_2 \blacktriangleright_{\overrightarrow{m_4}} \sigma_5 \to \sigma_6}{\begin{array}{l} \left(\overset{\sigma_1^\circ}{\Rightarrow}_{m_1}\dot{e}\overset{\sigma_2^\star}{\Rightarrow}\right) \lhd_{m_2} \left(\overset{\sigma_3^\circ}{\Rightarrow}_{m_3}e^{\Rightarrow}\right)\overset{\sigma_4^\circ}{\Rightarrow} \\ \Vdash\!\!\!\rightarrow \left(\overset{\sigma_1^\circ}{\Rightarrow}_{m_1}\dot{e}\overset{\sigma_2^\bullet}{\Rightarrow}\right) \lhd_{m_4} \left(\overset{\sigma_5^\star}{\Rightarrow}_{m_3}e^{\Rightarrow}\right)\overset{\sigma_6^\star}{\Rightarrow} \end{array}}$$

StepAsc

$$\frac{}{\begin{array}{l} \left(\overset{\sigma_1^\circ}{\Rightarrow}_{m}\dot{e}\right) \,:\, \tau^\star\overset{\sigma_2^\circ}{\Rightarrow} \\ \Vdash\!\!\!\rightarrow \left(\overset{\tau^\star}{\Rightarrow}_{m}\dot{e}\right) \,:\, \tau^\bullet\overset{\tau^\star}{\Rightarrow} \end{array}}$$

StepAp propagates a newly synthesized type from the function position of an application, with the same matched arrow type premise as in MALC. The resulting analyzed type for the body and synthesized type for the application are added to the frontier. StepAsc propagates changes from an ascribed type, with the new type synthesized by the form and analyzed in the body.

5.5 Generalized Approach

Although our presentation of Incremental MALC is given in terms of a minimal simply typed language, the ideas are applicable to a broader class of language features. The ideas of this section can be used to incrementalize any local flow of information through the syntax tree, and the ideas of the following section extend this to flow along binders governed by lexical scope. The approach can be applied to any language, but will not provide incrementalization for other kinds of computations. For example, the approach does not provide incrementalized type consistency checking for the simply typed language, nor does it provide incrementalized type substitution for System F or an incrementalized unification algorithm for unification-based type systems.

The general schema is as follows: the static semantics of each syntactic form in the MALC can be expressed as a function that determines each "output" of the node (the analyzed types of its children, its synthesized type, and its marks) in terms of the "inputs" of the node (its types in the surface syntax, its analyzed type, and the synthesized types of its children). These input and output

α	$\in$	Type Variable	
$\alpha?$	$\in$	Type Binding	$? \mid \alpha$
τ	$\in$	BareType	$\ldots \mid \alpha \mid \forall\alpha?.\ \tau$
$\check{\tau}$	$\in$	MarkedType	$\ldots \mid \alpha_m \mid \forall\alpha?.\ \check{\tau}$
Γ	$\in$	Context	$\ldots \mid \Gamma,\ \alpha$
$\check{\sigma}$	$\in$	MarkedTypeOpt	$\square \mid \check{\tau}$
e	$\in$	BareExp	$\ldots \mid \Lambda(\alpha?).\ e \mid e \lhd [\tau]$
$\check{\dot{e}}$	$\in$	MarkedConExp	$\ldots \mid \Lambda(\alpha?)_m.\ \check{\dot{e}} \mid \check{\dot{e}} \lhd_m [\check{\tau}^\circ]$
$\dot{e}$	$\in$	ConExp	$\ldots \mid \Lambda(\alpha?)_m.\ \dot{e} \mid \dot{e} \lhd_m [\check{\tau}^\circ]$
α	$\in$	SimpleAction	$\ldots \mid$ WrapTypFun $\mid$ WrapTypAp $\mid$ SetTypArg(τ)

Fig. 7. Polymorphism Syntax Extension

positions are ordered according to the direction in which information can flow: first the form's types in the surface syntax, then annotated types ordered by the left-to-right syntax order, and finally the form's marks. Action application always dirties the types inside or immediately flowing into the edited area, and update propagation steps always recompute the immediate consequences of a dirty type, clean the type, and then dirty the types changed by the update.

This schema is informal, but it enabled us to easily extend the system with additional language features in the mechanization, including products and System F style polymorphism. Language features such as polymorphic and dependent types contain substantial type-level computations that this approach cannot incrementalize, therefore will require additional techniques to achieve full incrementalization.

To illustrate the extensibility of the theory, here we include the additional grammar and inference rules for System F-style polymorphism in Figures 7-10. Types are extended with type variables and universal types, and expressions are extended with type function abstractions and applications. It is necessary to introduce marked types as distinct from bare types, and a type marking judgment. The type marking judgment places marks on free type variables and must be added as a premise to the marking rules for ascriptions and function abstractions (included in the appendix). The marking, action performance, and update propagation rules for type function abstractions and applications are analogous to ordinary abstractions and applications, utilizing a matched universal type judgment, a type variable update judgment, and a TypFunSyn function, which are all analogous to their simply typed counterparts. Type-level binding structure is considered up to alpha equivalence.

5.6 Agda Mechanization

All definitions, lemmas, theorems, and proofs of Incremental MALC, including those referenced in this section, have been mechanically proven in the Agda proof assistant. The mechanization includes all constructs of this section, including System F-style polymorphism, as well as product types. One difference between the written theory and the mechanization is that types are edited with fine-grained actions, like terms, instead of with coarse-grained actions such as SetAsc(τ). This simplifies the extension to polymorphic features. The mechanization also does not define the marked programs of Section 3, instead identifying them with the equivalent set of quiescent incremental programs. This choice frees us from needing two almost identical definitions of the central data structure of the theory. Termination is directly shown by proving that the inverse of the update propagation step relation is well-founded, which is then used to prove the given form of the theorem. The mechanization does not include the definitions of the total side condition functions, such as consistency and the matched arrow judgment, which are instead postulated.

$$\boxed{\Gamma \vdash \tau \rightsquigarrow \check{\tau}}$$

$$\boxed{\Gamma \vdash \check{\tau} \Rightarrow e \rightsquigarrow {}^{\Rightarrow}\check{e}}$$

$$\cdots \qquad \textsc{MarkTypVar}\ \frac{\alpha_m \in \Gamma}{\Gamma \vdash \alpha \rightsquigarrow \alpha_m} \qquad \textsc{MarkForall}\ \frac{\alpha?,\ \Gamma \vdash \tau \rightsquigarrow \check{\tau}}{\Gamma \vdash \forall \alpha?.\ \tau \rightsquigarrow \forall \alpha?.\ \check{\tau}}$$

$$\textsc{MarkAnaTypFun}\ \frac{\check{\tau}_1 \blacktriangleright^{\forall}_{m} \forall \alpha?.\ \check{\tau}_2 \qquad \Gamma,\ \alpha? \vdash \check{\tau}_2 \Rightarrow e \rightsquigarrow {}^{\Rightarrow}\check{e}}{\Gamma \vdash \check{\tau}_1 \Rightarrow \Lambda(\alpha?).\ e \rightsquigarrow \overset{\check{\tau}_1}{\Rightarrow}_{\checkmark} \Lambda(\alpha?)_m.\ {}^{\Rightarrow}\check{e} \overset{\square}{\Rightarrow}}$$

$$\boxed{\Gamma \vdash e \rightsquigarrow \check{e}^{\Rightarrow}}$$

$$\textsc{MarkSynTypFun}\ \frac{\Gamma,\ \alpha? \vdash e \rightsquigarrow \dot{\check{e}} \overset{\check{\tau}}{\Rightarrow}}{\Gamma \vdash \Lambda(\alpha?).\ e \rightsquigarrow \Lambda(\alpha?)_{\checkmark}.\ \left(\overset{\square}{\Rightarrow}_{\checkmark} \dot{\check{e}} \overset{\check{\tau}}{\Rightarrow}\right) \overset{\forall \alpha?.\ \check{\tau}}{\Rightarrow}}$$

$$\textsc{MarkTypAp}\ \frac{\Gamma \vdash e \rightsquigarrow \dot{\check{e}} \overset{\check{\tau}_1}{\Rightarrow} \quad \check{\tau}_1 \blacktriangleright^{\forall}_{m} \forall \alpha?.\ \check{\tau}_2 \qquad \Gamma \vdash \tau \rightsquigarrow \check{\tau}_3 \quad [\check{\tau}_3/\alpha?]\check{\tau}_2 = \check{\tau}_4}{\Gamma \vdash e \lhd [\tau] \rightsquigarrow \left(\overset{\square}{\Rightarrow}_{\checkmark} \dot{\check{e}} \overset{\check{\tau}_1}{\Rightarrow}\right) \lhd_m [\check{\tau}_3] \overset{\check{\tau}_4}{\Rightarrow}}$$

Fig. 8. Polymorphism Marking

$$\boxed{\Gamma \vdash \tau \overset{\alpha}{\longrightarrow} \check{\tau}}$$

$$\textsc{ActWrapTypFun}\ \frac{}{\Gamma \vdash \dot{e} \overset{\check{\sigma}^{\circ}}{\Rightarrow} \xrightarrow{\text{WrapFun}} \Lambda(?)_{\checkmark}.\ \left(\overset{\square^{\star}}{\Rightarrow}_{\checkmark} \dot{e} \overset{\check{\sigma}^{\bullet}}{\Rightarrow}\right) \overset{\square^{\star}}{\Rightarrow}}$$

$$\textsc{ActSetTypArg}\ \frac{\Gamma \vdash \tau \rightsquigarrow \check{\tau}_2}{\Gamma \vdash {}^{\Rightarrow}e \lhd_m [\check{\tau}_1^{\circ}] \overset{\check{\sigma}^{\circ}}{\Rightarrow} \xrightarrow{\text{SetTypArg}(\tau)} {}^{\Rightarrow}e \lhd_m [\check{\tau}_2^{\star}] \overset{\check{\sigma}^{\circ}}{\Rightarrow}}$$

$$\textsc{ActUnwrapTypFun}\ \frac{\alpha?_{m_3} \in \Gamma \qquad [\![\alpha?_{m_3}/\alpha?]\!] e^{\Rightarrow} = \dot{e} \overset{\check{\sigma}_3^{\circ}}{\Rightarrow}}{\Gamma \vdash \Lambda(\alpha?)_{m_1}.\ \left(\overset{\check{\sigma}_1^{\circ}}{\Rightarrow}_{m_2} e^{\Rightarrow}\right) \overset{\check{\sigma}_2^{\circ}}{\Rightarrow} \xrightarrow{\text{Unwrap(One)}} \dot{e} \overset{\check{\sigma}_3^{\star}}{\Rightarrow}}$$

$$\textsc{ActUnwrapTypAp}\ \frac{}{\Gamma \vdash \left(\overset{\check{\sigma}_1^{\circ}}{\Rightarrow}_{m_1} \dot{e} \overset{\check{\sigma}_2^{\circ}}{\Rightarrow}\right) \lhd_{m_2} [\check{\tau}^{\circ}] \overset{\check{\sigma}_3^{\circ}}{\Rightarrow} \xrightarrow{\text{Unwrap(One)}} \dot{e} \overset{\check{\sigma}_2^{\star}}{\Rightarrow}}$$

Fig. 9. Polymorphism Action Performance

$$\boxed{{}^{\Rightarrow}e \Vdash\!\!\!\rightarrow {}^{\Rightarrow}e}$$

$$\textsc{StepAnaTypFun}\ \frac{\check{\sigma}_1 \blacktriangleright^{\forall}_{m_3} \forall \alpha?.\ \check{\sigma}_5 \qquad \text{TypFunSyn}(\check{\sigma}_1, \check{\sigma}_3) = \check{\sigma}_6}{\begin{array}{c}\overset{\check{\sigma}_1^{\star}}{\Rightarrow}_{m_1} \Lambda(\alpha?)_{m_2}.\ \left(\overset{\check{\sigma}_2^{\circ}}{\Rightarrow}_{m_4} \dot{e} \overset{\check{\sigma}_3^{\circ}}{\Rightarrow}\right) \overset{\check{\sigma}_4^{\circ}}{\Rightarrow} \\ \Vdash\!\!\!\rightarrow \overset{\check{\sigma}_1^{\bullet}}{\Rightarrow}_{\checkmark} \Lambda(\alpha?)_{m_3}.\ \left(\overset{\check{\sigma}_5^{\star}}{\Rightarrow}_{m_4} \dot{e} \overset{\check{\sigma}_3^{\circ}}{\Rightarrow}\right) \overset{\check{\sigma}_6^{\star}}{\Rightarrow}\end{array}}$$

$$\textsc{StepSynTypFun}\ \frac{\text{TypFunSyn}(\check{\sigma}_1, \check{\sigma}_2) = \check{\sigma}_4}{\begin{array}{c}\overset{\check{\sigma}_1^{\circ}}{\Rightarrow}_{m_1} \Lambda(\alpha?)_{m_2}.\ \left(\overset{\square^{\circ}}{\Rightarrow}_{m_4} \dot{e} \overset{\check{\sigma}_2^{\star}}{\Rightarrow}\right) \overset{\check{\sigma}_3^{\circ}}{\Rightarrow} \\ \Vdash\!\!\!\rightarrow \overset{\check{\sigma}_1^{\circ}}{\Rightarrow}_{m_1} \Lambda(\alpha?)_{m_2}.\ \left(\overset{\square^{\circ}}{\Rightarrow}_{\checkmark} \dot{e} \overset{\check{\sigma}_2^{\bullet}}{\Rightarrow}\right) \overset{\check{\sigma}_4^{\star}}{\Rightarrow}\end{array}}$$

$$\boxed{e^{\Rightarrow} \Vdash\!\!\!\rightarrow e^{\Rightarrow}}$$

$$\textsc{StepTypApFun}\ \frac{\check{\sigma}_2 \blacktriangleright^{\forall}_{m_3} \forall \alpha?.\ \check{\sigma}_4 \qquad [\check{\tau}/\alpha?]\check{\sigma}_4 = \check{\sigma}_5}{\begin{array}{c}\left(\overset{\check{\sigma}_1^{\circ}}{\Rightarrow}_{m_1} \dot{e} \overset{\check{\sigma}_2^{\star}}{\Rightarrow}\right) \lhd_{m_2} [\check{\tau}^{\circ}] \overset{\check{\sigma}_3^{\circ}}{\Rightarrow} \\ \Vdash\!\!\!\rightarrow \left(\overset{\check{\sigma}_1^{\circ}}{\Rightarrow}_{m_1} \dot{e} \overset{\check{\sigma}_2^{\bullet}}{\Rightarrow}\right) \lhd_{m_3} [\check{\tau}^{\circ}] \overset{\check{\sigma}_5^{\star}}{\Rightarrow}\end{array}}$$

$$\textsc{StepTypApArg}\ \frac{\check{\sigma}_2 \blacktriangleright^{\rightarrow}_{m_4} \check{\sigma}_5 \to \check{\sigma}_6}{\begin{array}{c}\left(\overset{\check{\sigma}_1^{\circ}}{\Rightarrow}_{m_1} \dot{e} \overset{\check{\sigma}_2^{\circ}}{\Rightarrow}\right) \lhd_{m_2} [\check{\tau}^{\star}] \overset{\check{\sigma}_3^{\circ}}{\Rightarrow} \\ \Vdash\!\!\!\rightarrow \left(\overset{\check{\sigma}_1^{\circ}}{\Rightarrow}_{m_1} \dot{e} \overset{\check{\sigma}_2^{\circ}}{\Rightarrow}\right) \lhd_{m_3} [\check{\tau}^{\bullet}] \overset{\check{\sigma}_5^{\star}}{\Rightarrow}\end{array}}$$

Fig. 10. Polymorphism Update Propagation

This is because all metatheorems in the development are independent of the behavior of the side condition functions, so the proofs are kept abstract.

6 Implementation

The formalism of Incremental MALC primarily supports reasoning about the correctness properties of edit actions and update propagation steps. This section describes Malcom, our implementation of Incremental MALC. Critically, Malcom's edit actions and update propagation steps never need to traverse the unchanged portions of the program, including to build type contexts or search scopes for variable occurrences.

6.1 Ordered Mutable Trees

In Malcom, terms are represented as mutable trees with additional pointers. Cursors are represented by pointers to AST nodes, and each node stores a pointer to its parent, allowing constant time cursor movement up and down the tree. Nodes also store types and marks; both are mutable as well. Dirtiness is represented implicitly, by membership in a global priority queue, described later.

Moreover, terms in Malcom are decorated with two *timestamps* (a, b), where a is the node's position in a pre-order traversal of the full program and b is the node's position in a post-order traversal of the full program. Thus, $a < b$. These two timestamps form an interval $[a, b]$ with the expected properties: a term's interval strictly contains all its children's intervals, and sibling intervals are disjoint and ordered. The reader can imagine these timestamps as recording the left-to-right order of MALC annotations. For binders, we will refer to these intervals as representing the binder's scope, since the intervals contain all terms below the binder.

These timestamps have two beneficial properties. First, term containment corresponds to interval containment, so Malcom can quickly test whether one term (for example, a binder) contains another (for example, a variable). Secondly, MALC update steps take place in timestamp order, with the analytic steps performed in pre-order and the synthetic steps performed in post-order, which makes timestamps useful for prioritizing update steps.

Timestamps are not simple integers; they are elements of a global order maintenance structure. An order maintenance structure is a finite, totally ordered collection of elements e that allows fast creation and comparison of elements. Concretely, it supports just two basic operations:

- Insert(e), which constructs a new element directly after e.
- Compare(e_1, e_2), which compares the given elements in the total order.

Critically, both order maintenance operations are fast—O(1) with a small constant—and the order is preserved even as more elements are added. Malcom's implementation of order maintenance is based on Bender et al. [6] and extends the minimal API above with a variant of Insert that returns a new element directly before, instead of after, another element.

6.2 Executing Edit Actions and Update Steps

Most update steps and edit actions, which simply add or remove tree nodes and update their marks, can be performed directly. However, operations that affect variables and binders are challenging. Edit actions like changing the type annotation on a binder, changing the binder name, or deleting a binder can affect the type of all variables bound by that binder, which can be arbitrarily far away.

6.2.1 Variable and Binder Pointers. Malcom thus adds additional pointers to the tree to track binding information. Each variable stores a pointer to its binding site (that is, to the abstraction term that introduces it), and each binding site stores an ordered set of pointers to its bound variables, represented by a splay tree ordered by pre-order time stamp. It is convenient to treat the root of the program as the “binding site” for free variables, so it also stores a separate set of free variables for each variable name.

These additional pointers make certain update steps faster. For example, edits to type annotations on binders (SetAnn(τ)) can update all uses of the bound variable by iterating through the set of variable pointers at that binder. However, the pointer sets must also be maintained as edits occur.

Malcom also maintains a global map from variable names to the ordered set of binders for that variable name, whose importance will be explained shortly. These ordered sets are also represented by splay trees ordered by pre-order timestamp. Splay nodes also store both the post-order timestamp of each binder and also the maximum post-order timestamp in the subtree of the splay tree rooted at that splay node. The splay tree operations are implemented so as to maintain this additional piece of data. When binders are inserted or deleted, these ordered sets are directly updated.

6.2.2 Edits to Variables. First, consider edits to variables. When a variable is deleted, it need only be removed from its binding site’s set of bound variables. Since the variable stores a pointer to this site, this can be done directly. However, when a variable expression is inserted, it is necessary to locate its binding site (or the root if it is free) and add pointers between the variable and this site.

To do so, we use the global variable-name-to-binder map to find the set of all binders for the inserted variable name. To account for shadowing, we need the lowest binder that contains the insertion site; this is the binder with the largest pre-order timestamp whose interval contains the insertion site’s interval. The splay tree allows Malcom to find this node in $O(\log n)$ time; Malcom also splays this node to the root of the splay tree to make later operations using this binder faster.

6.2.3 Edits to Binders. Next, consider edits to binders. When a binder is created in the program, such as filling the empty binder of a function abstraction with a variable name, Malcom must check whether that binder shadows some outer binding and, if so, update all re-bound variables. Identifying the outer binding uses the same splay tree lookup as variable insertion. That binder has an ordered set of pointers to its bound variables; Malcom must determine which of these bound variables are now instead bound to the new binder.

The new binder is a descendant of the outer binder, so its scope is some subinterval of the outer binder’s scope. Therefore, Malcom needs to split the outer binder’s scope into three segments—an initial segment below only the outer binder, a middle segment below the new binder, and a final segment also only below the outer binder. To do so, it first considers the pre-order timestamp of the new binder, and splays that to the root of the outer binder’s splay tree. The root and the left subtree now represent the initial segment. The right subtree is removed and split again, on the new binder’s post-order timestamp; the right subtree of the result is now the final segment. The remaining middle segment now contains all bound variables of the new binder; it becomes the new binder’s ordered set of bound variables. Then the initial and final segments are joined (an $O(\log n)$ operation on splay trees) and become the outer binder’s new, smaller set of bound variables.

Deleting a binder is similar, but in reverse. The outer binder is found, and its split tree is split at the deleted binder’s pre-order timestamp. The two halves become the initial and final segments, joined to the deleted binder’s set of bound variables. The resulting larger set becomes the outer binder’s new set of bound variables. In case of both insertion and deletion, the variable’s global set of binders is also updated.

While complex, this splay-tree-based algorithm allows for algorithmically efficient handling of binders without the need to traverse the program to search for other binders or bound variables.

6.3 Prioritizing Update Steps

Another challenge is actually locating possible update steps. Malcom achieves this by maintaining a priority queue of all locations where update steps are possible (all dirtied types), represented as pointers into the AST. Each update step, then, simply involves popping a dirtied type location from the priority queue, performing the corresponding update step, and inserting any newly dirtied type locations into the priority queue. The action performance and update propagation step rules of Incremental MALC have been crafted to ensure that they can be implemented this way. For example, the premise to each update propagation step rule requires exactly one type to be dirty, and the conclusion cleans this type while dirtying some other types. In Malcom, this corresponds to popping an update location from the queue, mutating local information in the program, and pushing some new set of update locations onto the queue.

The priority queue is ordered using the term's timestamps. For new analyzed types or types in the surface syntax, the priority is the node's pre-order timestamp, since analysis steps are performed pre-order. For new synthesized types, it is the post-order timestamp. Since Incremental MALC is non-deterministic, the order *does not affect correctness*. However, the chosen order is more efficient in many cases. Since bidirectional type checking moves "left to right" through the program, and the chosen order is also a "left to right" order, Malcom will also push terms with larger timestamps than the ones it just popped. This means that Malcom tends to perform upstream updates before downstream updates, so as to avoid duplicate work.

Finally, there's one last edge case: term deletion. In the calculus, deletion is trivial because all data is local. However, with the global update queue, care must be taken when deleting subexpressions that contain update locations. When a subexpression is deleted, it is traversed, with each AST node within being marked as deleted. When an update location is popped form the queue, it is first checked that the location has not been deleted. If it has been, it is skipped, and popping continues. This is the only operation in Malcom that traverses an entire subexpression, but this cost is still proportional to the size of the edit and should be small for most reasonable editing patterns. If desired, the priority queue could be implemented by a splay tree, which would allow for easy deletion of pending updates to deleted terms. However, we found that a standard min-heap priority queue was faster and did not use this technique in our implementation.

6.4 Unchanged Type Optimization

As defined in Incremental MALC, actions and update propagation steps dirty all types that could have potentially changed, even if they happen to remain the same. A simple optimization is to forgo dirtying types when unchanged. This cannot be applied in all cases, such as the dirtying of the analyzed type where an action is performed. In this cases, it is because the expression changes, not the type, that the analyzed type must be propagated. This optimization is, however, applicable to all update propagation steps, as they do not change the form of the expression.

This optimization is implemented in Malcom for all update propagation steps. However, this requires comparing dirtied types for structural equality, which is linear in the size of the type in our implementation. This should not be an issue for typical programs, and could be addressed if desired by hash-consing type syntax trees and comparing hashes.

6.5 Language Workbench

We have implemented Malcom, as described above, as a language workbench written in OCaml. It implements the data structures described above, including order maintenance, splay trees, variable and binder pointers, and the name-to-binder map. Traversal of the entire program is never necessary.

We also wrote a simple interface for Malcom that renders the program textually in a web browser. It maintains one cursor. There is a button for each edit action and each cursor movement. There is also an "update step" button that triggers the first update step and an "all update steps" button that triggers update steps until the program is quiescent. Synthesized and analyzed typing information is hidden, except when they are members of the update propagation frontier, which allowed us

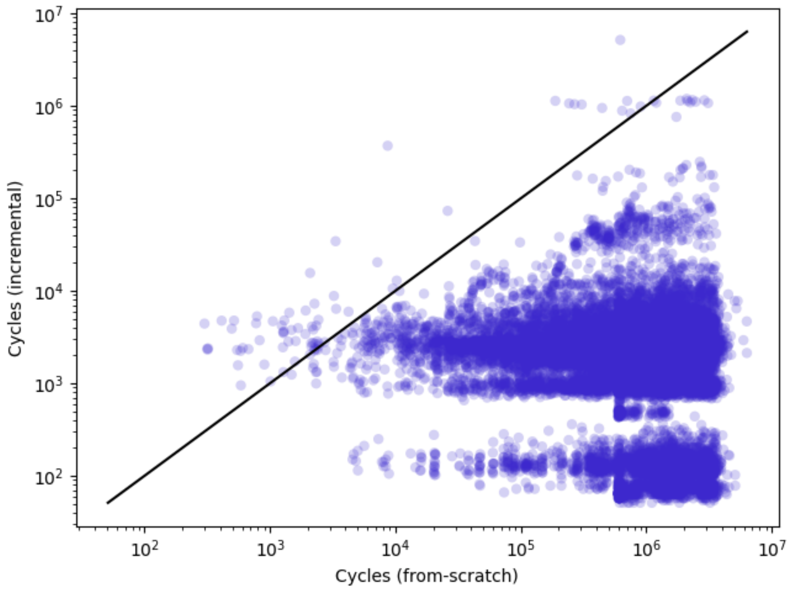

Fig. 11. Incremental vs. from-scratch editing times

to observe update propagation and debug our implementation. Note that the web interface is for debugging only, and while it updates type information incrementally, the actual interface is rendered from scratch at each step. We leave incremental rendering of the incrementally-computed type information for future work. We have extensively tested Malcom to ensure that the incremental output is always consistent with the from-scratch output.

7 Evaluation

In this section we present our evaluation of the efficiency of Malcom relative to from-scratch type checking.

7.1 Benchmark

We time Malcom on a synthesized edit trace constructed to test multiple aspects of Malcom. The first stage of the edit trace builds the program to a substantial size by constructing 100 nested copies of the merge sort algorithm. Each layer consists of a split function, a merge function, and a sort function. Each instance of the two helper functions, merge and split, is bound to a unique numbered identifier (`split_n` and `merge_n` for each layer `n`). Each sort function, on the other hand, is bound to the same identifier (`mergesort`). To capture long-distance binding patterns, each implementation of sort chooses which copy of split and which copy of merge to use uniformly at random from among those in scope. This tests two aspects of Malcom's handling of bindings.

(1) **Shadowing**. Each `mergesort` shadows the one before it, testing Malcom's ability to find ancestor binders efficiently.
(2) **Name Reuse**. The definitions of `split_n`, `merge_n`, and `mergesort` each use common local variable names, such as `x`. Thus, those variable names are bound in multiple non-overlapping scopes, and Malcom must bind each occurrence to the correct binder.

The edit trace builds this tower by inserting constructors at leaves of the program sketch until complete. After each node is constructed, its children are constructed in a random order, proceeding recursively. This randomness ensures that different patterns of variable interactions are tested: for example, a variable binding might be inserted into an abstraction after its body has been constructed, capturing the free occurrences in the body.

After the construction phase of the edit trace, random edits are applied throughout the program. Each of these consists of moving to a uniformly random location, applying a small change, and reverting the change. This is repeated 500 times. The changes include:

(1) Deleting a leaf node and inserting another in its place.
(2) Replacing a binder with another.
(3) Wrapping a constructor around a subterm.
(4) Alternatively, for constructors with one child subexpression (for which unwrapping does not delete subexpressions), unwrapping the constructor.

These edits test deletion, insertion, wrapping, and unwrapping, including updating types and binders. They also can introduce errors, demonstrating Malcom's incremental error marking ability.

7.2 Results

For each non-move edit, we measure the time it takes to perform the action and propagate updates until quiescent. Timing is done with `rdtsc`. For a baseline, the same edit is performed on a bare syntax tree represented with a zipper data structure, and is type checked from scratch according to the ordinary MALC marking procedure. The total time to edit and type check is compared between the incremental and from-scratch analyses. Note that action performance can be slower for Malcom than for the ordinary zipper representation, since Malcom computes changes to bindings at edit time. Figure 11 show a scatter plot comparing these times (in cycles) for Malcom and the from-scratch analysis. Each data point is one non-move edit to the program. Points above the diagonal (shown in black) are edits where the from-scratch analysis is faster than the incremental one, and below the diagonal is the converse. Almost all points are below the diagonal, meaning that incrementality provides a speedup in almost all cases. Many points are far below the diagonal, with incrementality providing one or more orders of magnitude of speedup. The data points are multi-modal, forming multiple horizontal bands. This indicates a fundamental algorithmic speedup: while the from-scratch analysis grows linearly in program size, empirically the incremental analysis shows hardly any growth. The horizontal bands are likely made up of different kinds of actions. The top-most band displays a slight positive trend, indicating cases where the incremental analysis does grow with the program size. This band likely contains edits that affect binding structure.

In total across all of these edits, Malcom provides whopping 275.96×speedup over from-scratch marking, demonstrating its substantial advantage over non-incremental analysis in providing continuous static feedback.

8 Related Work

There is a large body of work on incremental computation [21, 36], ranging from general frameworks such as Self-Adjusting Computation (SAC) [2], Adaption [16, 17], and incremental Datalog engines [49] to domain-specific approaches for lists [34], databases [24], and web browsers [14].

Order maintenance data structures, first introduced by [10], are a common data structure across these approaches, appearing in various forms in both general frameworks like SAC [3] and domain-specific systems like web browser layout engines [19].

When considering the specific problem tackled in this paper of incrementally maintaining type information in response to program edits, there have been prior efforts of both varieties as well. Our approach is a domain-specific approach targeted very specifically to bidirectional type checking with total error localization, building on recent advances, most notably the marked lambda calculus [50] which was described in Section 2. Fundamental to our approach is the use of a structure editor calculus to represent changes. We are most directly inspired by Hazelnut [28], which specified a bidirectionally typed structure editor calculus which was able to update type error marks locally at the cursor without needing recomputation. However, Hazelnut simply leaves edits that require distant changes to error marks undefined, making it impractical for real editing. Hazel, which was originally based on Hazelnut, now uses an approach based on the MLC, and we plan to integrate the approach from this paper into a future version of Hazel.

The small step approach to typing appears in prior work [20, 41], in which typing is presented as a rewrite system and is analyzed using term-rewriting theory. The use of small steps to propagate types is similar to Incremental MALC, but without birdirectionality or total error localization.

Szabó et al. [42] and Pacak et al. [30] take a more general approach by translating typing rules to a Datalog program whose database of derived semantic facts can be incrementally updated as the facts that encode the program are updated. This generic approach is worthy of continued investigation, but is limited by the generality of the incremental algorithm for the chosen implementation of Datalog. One particular challenge is accounting for bindings in a way that leverages the incrementality of the solver. Pacak et al. [30] uses co-contextual typing [12], in which binding information is propagated bottom up rather than top down, to improve incrementality. This approach, while fine grained enough to capture individual binding updates rather than whole-context changes, still requires traversing program spines, incurring linear time performance penalties.

Demers et al. [9] discusses incremental evaluation of attribute grammars, proposing two different approaches. As with Datalog, it is possible to encode many type systems as attribute grammars, and indeed the setting in this paper is the Cornell Program Synthesizer [43], one of the earliest structure editors and a pioneer in live semantic analysis. It may be that a generalized version of our approach would start to look like an incremental attribute grammar system, given analogies between synthesized and analyzed attributes and synthesized and analyzed types. We are not aware of a contemporary implementation of these ideas.

Our approach in this paper is distinctive relative to these related approaches in that we start by approaching the problem from type-theoretic first principles, developing a formal semantics for type information propagation and proving its metatheoretic properties, then implementing it in a highly specialized manner targeted to the problem of incremental bidirectional typing. We believe it will be fruitful to continue to compare generic approaches to our more direct approach, and hope that the novel benchmark task developed here will be ported to other systems and lead to productive competition (and even more realistic or punishing benchmarks).

In addition to approaches leveraging incrementality derived from edits, there are a number of approaches that rely on memoization or caching to improve the performance of language implementations. This includes a large body of work on incremental build systems [1, 26], which are in common use in industry. These systems generally operate at a coarse granularity, rebuilding entire compilation units (e.g. modules or packages) after changes. In contrast, our approach operates at a fine-grained level, on individual expressions. Consequently, our approach is more suitable for live programming environments where edits may occur many times per second.

This family of approaches also includes the system of Wachsmuth et al. [44], which develops a name resolution and type analysis engine using dependent "tasks," which each capture one step of analysis. This approach is language-independent but again less fine-grained than our approach, operating primarily at the file level in the implemented system. The priority queue maintaining the update propagation frontier in Malcom is similar to a task engine.

Busi et al. [7] specify memoized versions of standard typing rules and implement this system to achieve speedups on various small functional and imperative benchmarks, but their approach sometimes requires traversing large parts of the program when a variable binding changes.

Aditya and Nikhil [4] present a system that incrementalizes the Hindley-Milner type inference algorithm based on a call-graph analysis. It specifically incrementalizes the unification phase of type checking, operating at the granularity of changes to declarations. Meertens [25] also presents an incremental system for unification-based type checking, which incrementally maintains the constraints generated by the program but does not incrementally solve them in general.

Incrementalization has also been applied to program analysis techniques, such as abstract interpretation [13, 23, 37, 38], symbolic execution [35, 45–47], intermediate representation [15], data flow for probabilistic programs [48], and analysis via encoding as a constraint satisfaction problem [27]. Existing techniques for incremental abstract interpretation could be applied to type inference, but it is not clear how they would handle type error localization, fine grained incremental analysis, or concurrent editing and analysis.

9 Discussion and Conclusion

In this work we provide a formal system and efficient implementation for maintaining type information in a simple editor calculus. To serve as the base theory, we introduce the marked and annotated lambda calculus (MALC), a novel variant of the marked lambda calculus.

Our incremental system is remarkably (pun intended) more efficient than from-scratch reanalysis. It contributes to the goal of live programming not just by being efficient, but by reducing the extent to which type checking blocks users from editing the program.

9.1 Future Work

Incremental MALC and Malcom present several opportunities for extension and improvement. One notable limitation of our approach is that although update propagation is less blocking than a batch analysis, it is still possible for some individual update propagation steps to take a long time. In particular, those that involve traversing every occurrence of a variable bound by a particular binder block editing for a time proportional to the number of bound variables. Although this would not seem to be much of a problem in practice, future work may refine the approach and address this possibility. One potential strategy is for the binder to track which of its variables have been updated with new information and update the variables one at a time during update propagation.

While Incremental MALC expresses highly incremental expression-level analysis, it does not provide an incremental solution for type-level computations. Consistency checks, for example, must be rerun in their entirety when only part of the input changes. Although the size of types might often remain modest compared with the size of the program, it may be valuable to extend Incremental MALC with incremental type-level operations in future work.

As described in Section 5.5, Incremental MALC as presented in this work fits within a more general schema that can accommodate additional language features. We leave it to future work to define, validate, and explore the expressivity of such a schema. We also leave to future work the scaling of these ideas up to real-world type systems.

Incremental MALC supports liveness for only type-based editor services. To achieve large scale live programming requires incrementalized theories for other editor services, such as evaluation. There is much future work to be done in developing such theories.

Data Availability Statement

This paper is accompanied by two artifacts: an Agda mechanization of the definitions and proofs for Incremental MALC (presented in Section 5) and an OCaml implementation and interactive workbench of the Malcom incremental type system (described in Section 6). Both are included in the accompanying artifact [32].

Acknowledgments

The authors would like to thank the anonymous referees at WITS 2025 and OOPSLA 2025 for their helpful feedback on this project. The authors would also like to thank Eric Zhao, Sebastian Erdweg, and Michael Arntzenius for their illuminating conversations related to this work, and thank Liam Mulcahy for his contributions to the accompanying workbench. We credit Kanat Tangwongsan, Yit Phang Khoo, and Matthew A. Hammer for their implementation of the order maintenance data structure. This work was partially funded by the National Science Foundation under Grant No. 2238744 and 2340192. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] 2018. salsa. https://github.com/salsa-rs/salsa.
[2] Umut A Acar. 2005. *Self-adjusting computation.* Carnegie Mellon University.
[3] Umut A. Acar, Guy E. Blelloch, and Robert Harper. 2002. Adaptive functional programming. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, John Launchbury and John C. Mitchell (Eds.). ACM, 247–259. https://doi.org/10.1145/503272.503296
[4] Shail Aditya and Rishiyur S. Nikhil. 1991. Incremental Polymorphism. In *Functional Programming Languages and Computer Architecture, 5th ACM Conference, Proceedings (Lecture Notes in Computer Science, Vol. 523)*, John Hughes (Ed.). Springer, 379–405. https://doi.org/10.1007/3540543961_19
[5] Alexander Bandukwala, Andrew Blinn, and Cyrus Omar. 2024. Toward a Live, Rich, Composable, and Collaborative Planetary Compute Engine. In *Workshop on Programming for the Planet (PROPL)*.
[6] Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. 2002. Two Simplified Algorithms for Maintaining Order in a List. In *Algorithms - ESA 2002, 10th Annual European Symposium, Proceedings (Lecture Notes in Computer Science, Vol. 2461)*, Rolf H. Möhring and Rajeev Raman (Eds.). Springer, 152–164. https://doi.org/10.1007/3-540-45749-6_17
[7] Matteo Busi, Pierpaolo Degano, and Letterio Galletta. 2019. Using Standard Typing Algorithms Incrementally. In *NASA Formal Methods - 11th International Symposium, NFM 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11460)*, Julia M. Badger and Kristin Yvonne Rozier (Eds.). Springer, 106–122. https://doi.org/10.1007/978-3-030-20652-9_7
[8] Sudarshan S Chawathe and Hector Garcia-Molina. 1997. Meaningful change detection in structured data. *ACM SIGMOD Record* 26, 2 (1997), 26–37.
[9] Alan J. Demers, T. Reps, and Tim Teitelbaum. 1981. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *ACM-SIGACT Symposium on Principles of Programming Languages*. https://api.semanticscholar.org/CorpusID:12044550
[10] Paul F Dietz. 1982. Maintaining order in a linked list. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*. 122–127.
[11] Jana Dunfield and Neel Krishnaswami. 2022. Bidirectional Typing. *ACM Comput. Surv.* 54, 5 (2022), 98:1–98:38. https://doi.org/10.1145/3450952
[12] Sebastian Erdweg, Oliver Bracevac, Edlira Kuci, Matthias Krebs, and Mira Mezini. 2015. A co-contextual formulation of type rules and its application to incremental type checking. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 880–897. https://doi.org/10.1145/2814270.2814277
[13] Noah Van Es, Maarten Vandercammen, and Coen De Roover. 2017. Incrementalizing Abstract Interpretation. In *Proceedings of the 16th edition of the BElgian-NEtherlands software eVOLution symposium (CEUR Workshop Proceedings, Vol. 2047)*, Serge Demeyer, Ali Parsai, Gulsher Laghari, and Brent van Bladel (Eds.). CEUR-WS.org, 31–35. https://ceur-ws.org/Vol-2047/BENEVOL_2017_paper_9.pdf
[14] Tali Garseil. 2010. How browsers work — taligarsiel.com. https://taligarsiel.com/Projects/howbrowserswork1.htm#Dirty_bit_system. [Accessed 05-11-2024].
[15] Vaidehi Ghime, Ankita Khadsare, Anushri Jana, and Bharti Chimdyalwar. 2022. IR Mapping: Intermediate Representation (IR) based Mapping to facilitate Incremental Static Analysis. In *ISEC 2022: 15th Innovations in Software Engineering Conference*, Saurabh Tiwari, Sanjay Chaudhary, Chanchal K. Roy, Meenakshi D'Souza, Richa Sharma, and Lov Kumar

(Eds.). ACM, 23:1–23:5. https://doi.org/10.1145/3511430.3511451

[16] Matthew A Hammer, Jana Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S Foster, Michael Hicks, and David Van Horn. 2015. Incremental computation with names. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 748–766.

[17] Matthew A. Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S. Foster. 2014. Adapton: composable, demand-driven incremental computation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 156–166. https://doi.org/10.1145/2594291.2594324

[18] Amelia Holcomb, Michael Dales, Patrick Ferris, Sadiq Jaffer, Thomas Swinfield, Alison Eyres, Andrew Balmford, David Coomes, Srinivasan Keshav, and Anil Madhavapeddy. 2023. A Case for Planetary Computing. *arXiv e-prints* (2023), arXiv–2303.

[19] Marisa Kirisame, Tiezhi Wang, and Pavel Panchekha. 2025. Spineless Traversal for Layout Invalidation. arXiv:2411.10659 [cs.PL] https://arxiv.org/abs/2411.10659

[20] George Kuan, David MacQueen, and Robert Bruce Findler. 2007. A Rewriting Semantics for Type Inference. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4421)*, Rocco De Nicola (Ed.). Springer, 426–440. https://doi.org/10.1007/978-3-540-71316-6_29

[21] Yanhong A. Liu. 2024. Incremental Computation: What Is the Essence? (Invited Contribution). In *Proceedings of the 2024 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation (PEPM 2024)*. Association for Computing Machinery, New York, NY, USA, 39–52. https://doi.org/10.1145/3635800.3637447

[22] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)* 10, 4 (2010), 1–15.

[23] Scott McPeak, Charles-Henri Gros, and Murali Krishna Ramanathan. 2013. Scalable and incremental software bug detection. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13*, Bertrand Meyer, Luciano Baresi, and Mira Mezini (Eds.). ACM, 554–564. https://doi.org/10.1145/2491411.2501854

[24] Frank McSherry, Derek Murray, Rebecca Isaacs, and Michael Isard. 2013. Differential dataflow. In *Proceedings of CIDR 2013* (proceedings of cidr 2013 ed.). https://www.microsoft.com/en-us/research/publication/differential-dataflow/

[25] Lambert G. L. T. Meertens. 1983. Incremental Polymorphic Type Checking in B. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, John R. Wright, Larry Landweber, Alan J. Demers, and Tim Teitelbaum (Eds.). ACM Press, 265–275. https://doi.org/10.1145/567067.567092

[26] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. 2018. Build systems à la carte. *Proc. ACM Program. Lang.* 2, ICFP, Article 79 (July 2018), 29 pages. https://doi.org/10.1145/3236774

[27] Rashmi Mudduluru and Murali Krishna Ramanathan. 2014. Efficient Incremental Static Analysis Using Path Abstraction. In *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8411)*, Stefania Gnesi and Arend Rensink (Eds.). Springer, 125–139. https://doi.org/10.1007/978-3-642-54804-8_9

[28] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: a bidirectionally typed structure editor calculus. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017*, Giuseppe Castagna and Andrew D. Gordon (Eds.). ACM, 86–99. https://doi.org/10.1145/3009837.3009900

[29] Cyrus Omar, Ian Voysey, Michael Hilton, Joshua Sunshine, Claire Le Goues, Jonathan Aldrich, and Matthew A. Hammer. 2017. Toward Semantic Foundations for Program Editors. In *2nd Summit on Advances in Programming Languages, SNAPL 2017, May 7-10, 2017, Asilomar, CA, USA (LIPIcs, Vol. 71)*, Benjamin S. Lerner and Shriram Krishnamurthi Rastislav Bodík (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 11:1–11:12. https://doi.org/10.4230/LIPICS.SNAPL.2017.11

[30] André Pacak, Sebastian Erdweg, and Tamás Szabó. 2020. A systematic approach to deriving incremental type checkers. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 127:1–127:28. https://doi.org/10.1145/3428195

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

[32] Thomas J. Porter, Marisa Kirisame, Ivan Wei, Pavel Panchekha, and Cyrus Omar. 2025. *Artifact for Incremental Bidirectional Typing via Order Maintenance*. https://doi.org/10.5281/zenodo.16922160

[33] Jacob Prinz, Henry Blanchette, and Leonidas Lampropoulos. 2025. Pantograph: A Fluid and Typed Structure Editor. *Proceedings of the ACM on Programming Languages* 9, POPL (2025), 802–831.

[34] William Pugh and Tim Teitelbaum. 1989. Incremental computation via function caching. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 315–328.

[35] Rui Qiu, Guowei Yang, Corina S. Pasareanu, and Sarfraz Khurshid. 2015. Compositional Symbolic Execution with Memoized Replay. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015*, Antonia Bertolino,

Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE Computer Society, 632–642. https://doi.org/10.1109/ICSE.2015.79

[36] Ganesan Ramalingam and Thomas Reps. 1993. A categorized bibliography on incremental computation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 502–510.

[37] Mamy Razafintsialonina, David Bühler, Antoine Miné, Valentin Perrelle, and Julien Signoles. 2025. Reusing Caches and Invariants for Efficient and Sound Incremental Static Analysis. In *39th European Conference on Object-Oriented Programming, ECOOP 2025 (LIPIcs, Vol. 333)*, Jonathan Aldrich and Alexandra Silva (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 28:1–28:29. https://doi.org/10.4230/LIPICS.ECOOP.2025.28

[38] Helmut Seidl, Julian Erhard, and Ralf Vogler. 2020. Incremental Abstract Interpretation. In *From Lambda Calculus to Cybersecurity Through Program Analysis - Essays Dedicated to Chris Hankin on the Occasion of His Retirement (Lecture Notes in Computer Science, Vol. 12065)*, Alessandra Di Pierro, Pasquale Malacaria, and Rajagopal Nagarajan (Eds.). Springer, 132–148. https://doi.org/10.1007/978-3-030-41103-9_5

[39] Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme and Functional Programming Workshop*.

[40] Jeremy G. Siek, Michael M. Vitousek, Matteo Cimini, and John Tang Boyland. 2015. Refined Criteria for Gradual Typing. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015 (LIPIcs, Vol. 32)*, Thomas Ball, Rastislav Bodik, Shriram Krishnamurthi, Benjamin S. Lerner, and Greg Morrisett (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 274–293. https://doi.org/10.4230/LIPICS.SNAPL.2015.274

[41] Aaron Stump, Garrin Kimmell, and Roba El Haj Omar. 2011. Type Preservation as a Confluence Problem. In *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications, RTA 2011 (LIPIcs, Vol. 10)*, Manfred Schmidt-Schauß (Ed.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 345–360. https://doi.org/10.4230/LIPICS.RTA.2011.345

[42] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: a DSL for the definition of incremental program analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, David Lo, Sven Apel, and Sarfraz Khurshid (Eds.). ACM, 320–331. https://doi.org/10.1145/2970276.2970298

[43] Tim Teitelbaum and Thomas Reps. 1981. The Cornell program synthesizer: a syntax-directed programming environment. *Commun. ACM* 24, 9 (1981), 563–573.

[44] Guido Wachsmuth, Gabriël D. P. Konat, Vlad A. Vergu, Danny M. Groenewegen, and Eelco Visser. 2013. A Language Independent Task Engine for Incremental Name and Type Analysis. In *Software Language Engineering - 6th International Conference, SLE 2013, Proceedings (Lecture Notes in Computer Science, Vol. 8225)*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.). Springer, 260–280. https://doi.org/10.1007/978-3-319-02654-1_15

[45] Guowei Yang, Sarfraz Khurshid, and Corina S. Pasareanu. 2013. Memoise: a tool for memoized symbolic execution. In *35th International Conference on Software Engineering, ICSE '13*, David Notkin, Betty H. C. Cheng, and Klaus Pohl (Eds.). IEEE Computer Society, 1343–1346. https://doi.org/10.1109/ICSE.2013.6606713

[46] Guowei Yang, Corina S. Pasareanu, and Sarfraz Khurshid. 2012. Memoized symbolic execution. In *International Symposium on Software Testing and Analysis, ISSTA 2012*, Mats Per Erik Heimdahl and Zhendong Su (Eds.). ACM, 144–154. https://doi.org/10.1145/2338965.2336771

[47] Guowei Yang, Suzette Person, Neha Rungta, and Sarfraz Khurshid. 2014. Directed Incremental Symbolic Execution. *ACM Trans. Softw. Eng. Methodol.* 24, 1 (2014), 3:1–3:42. https://doi.org/10.1145/2629536

[48] Jieyuan Zhang, Yulei Sui, and Jingling Xue. 2017. Incremental Analysis for Probabilistic Programs. In *Static Analysis - 24th International Symposium, SAS 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10422)*, Francesco Ranzato (Ed.). Springer, 450–472. https://doi.org/10.1007/978-3-319-66706-5_22

[49] David Zhao, Pavle Subotic, Mukund Raghothaman, and Bernhard Scholz. 2021. Towards Elastic Incrementalization for Datalog. In *Proceedings of the 23rd International Symposium on Principles and Practice of Declarative Programming (PPDP '21)*. Association for Computing Machinery, New York, NY, USA, Article 20, 16 pages. https://doi.org/10.1145/3479394.3479415

[50] Eric Zhao, Raef Maroof, Anand Dukkipati, Andrew Blinn, Zhiyi Pan, and Cyrus Omar. 2024. Total Type Error Localization and Recovery with Holes. *Proc. ACM Program. Lang.* 8, POPL (2024), 2041–2068. https://doi.org/10.1145/3632910