Using Standard Typing Algorithms Incrementally *

Matteo Busi[1][0000−0002−5557−8139], Pierpaolo Degano[1][0000−0002−8070−4838], and Letterio Galletta[2][0000−0003−0351−9169]

[1] Dipartimento di Informatica, Università di Pisa, Pisa, Italy
{matteo.busi,degano}@di.unipi.it
[2] IMT School for Advanced Studies, Lucca, Italy
letterio.galletta@imtlucca.it

Abstract. Modern languages are equipped with static type checking/inference that helps programmers to keep a clean programming style and to reduce errors. However, the ever-growing size of programs and their continuous evolution require building fast and efficient analysers. A promising solution is *incrementality*, aiming at only re-typing the *diffs*, i.e. those parts of the program that change or are inserted, rather than the entire codebase. We propose an algorithmic schema that drives an incremental usage of existing, standard typing algorithms with no changes. Ours is a *grey-box* approach: just the shape of the input, that of the results and some domain-specific knowledge are needed to instantiate our schema. Here, we present the foundations of our approach and the conditions for its correctmess. We show it at work to derive two different incremental typing algorithms. The first type checks an imperative language to detect information flow and non-interference, and the second infers types for a functional language. We assessed our proposal on a prototypical implementation of an incremental type checker. Our experiments show that using the type checker incrementally is (almost) always rewarding.

1 Introduction

Most of the modern programming languages are equipped with mechanisms for checking or inferring types. Such static analyses prescribe programmers a clean programming style and help them to reduce errors. The ever-growing size of programs requires building fast and efficient analyzers. This quest becomes even more demanding because many companies are recently adopting development methodologies that advocate a continuous evolution of software, e.g. *perpetual development model* [4]. In such a model a shared code base is altered by many programmers submitting small code modifications (*diffs*). Software systems are no longer monolithic pieces of code, to which only new components/modules can be compositionally added, rather their components grow and change incrementally.

* The first two authors have been partially supported by U. Pisa project PRA_2018_66 *DECLware: Declarative methodologies for designing and deploying applications*. The last author is supported by IMT project *PAI VeriOSS*.

Consequently, analyses and verification should only consider the *diffs*, rather than the entire codebase. As recently observed by [7], it becomes crucial defining algorithms that require an amount of work on the size of the *diffs* instead of the whole codebase. The idea is to store summaries of the analysis results for program components, and to only reanalyze changed parts, re-using the cached summaries. Here we formalise this idea focussing on type systems.

The literature reports on some techniques, briefly surveyed below, which introduce *new* typing algorithms that work incrementally. Instead, we propose a method that takes an *existing* typing algorithm and *uses it incrementally*, without re-doing work already done, but exploiting available summaries, through caching and memoization. An advantage of our proposal is that it consists of an algorithmic schema *independent* of any specific language and type system. In addition, we put forward a mild condition on the summaries that guarantees that the results of incremental typing match those of the original algorithm.

Roughly, our schema works as follows. We start from the abstract syntax tree of the program, where each node is annotated with the result R provided by the original typing algorithm $\mathcal{A}$. We build then a cache, containing for each subterm t the result R and other relevant contextual information needed by $\mathcal{A}$ to type t (typically a typing environment binding the free variables of t). When the program changes, its annotated abstract syntax tree changes accordingly and typing the subterm associated with the changed node is done incrementally, by reusing the results in the cache whenever possible and by suitably invoking $\mathcal{A}$ upon need. Clearly, the more local the changes, the more information is reused.

Technically, our proposal consists of a set of rule schemata that drive the usage of the cache and of the original algorithm $\mathcal{A}$, as sketched above. Actually, the user has to define the shape of caches and to instantiate a well-confined part of the rule schemata. If the instantiation meets an easy-to-check criterion, the typing results of $\mathcal{A}$ and of the incremental algorithm are guaranteed to be coherent, i.e. the incremental algorithm behaves as the non-incremental one. All the above provides us with the guidelines to develop a framework that makes incremental the usage of a given typing algorithm.

Summing up, the main contributions of this paper include:

- a parametric, language-independent algorithmic schema that uses an existing typing algorithm $\mathcal{A}$ incrementally (Section 3);
- a formalisation of the steps that instantiate the schema and yield the incremental version of $\mathcal{A}$: the resulting typing algorithm only types the *diffs* and those parts of the code affected by them (Section 3);
- a characterisation of the rule format of standard typing algorithms in terms of two functions *tr* and *checkJoin* (Section 3);
- a theorem that under a mild condition guarantees the coherence of results between the original algorithm and its incremental version (Section 3);
- the instantiation of the schema for type checking and type inference algorithm for an imperative (Section 4) and a functional language (Section 5);

- a prototype of the incremental version of the type checker for MinCaml [20],[3] showing that implementing the schema is doable (Section 6); and
- experimental results showing that the cost of using the type checker incrementally depends on the size of *diffs*, and its performance increases as these become smaller (Section 6).

All the proofs of our theorems, some additional material and the tables with the experimental results on the time and space overheads are in the extended version available online [3].

Related work. To the best of our knowledge, the literature has some proposals for incrementally typing programs. However, these approaches heavily differ from ours, because all of them propose a *new* incremental algorithm for typing, while we *incrementally* use *existing* algorithms as they are. Additionally, none of the approaches surveyed below use a uniform characterisation of type judgements as we do through the metafunctions *tr* and *checkJoin*.

Meertens [13] proposes an incremental type checking algorithm for the language `B`. Johnson and Walz [9] treat incremental type inference, focussing on identifying where type errors precisely arise. Aditya and Nikhil [1] propose an incremental Hindley/Milner type system supporting incremental type checking of top-level definitions. Our approach instead supports incremental type-checking for all kinds of expressions, not only the top-level ones. Miao and Siek [14] introduce an incremental type checker leveraging the fact that, in multi-staged programming, programs are successively refined. Wachsmuth et al. [23] propose a task engine for type checking and name resolution: when a file is modified a task is generated and existing (cached) results are re-used where possible. The proposal by Erdweg et al. [5] is the most similar to ours, but, given a type checking algorithm, they describe how to obtain a *new* incremental algorithm. As in our case, they decorate an abstract syntax tree with types and typing environments, represented as sets of constraints, to be suitably propagated when typing. In this way there is no need of dealing with top-down context propagation while types flow bottom-up. Recently, Facebook released Pyre [6] a scalable and incremental type checker specifically designed for `Python`.

Incrementality has also been studied for static analysis other than typing. IncA [21] is a domain-specific language for the definition of incremental program analyses, which represents dependencies among the nodes of the abstract syntax tree of the target program as a graph. Infer [8] uses an approach similar to ours in which analysis results are cached to improve performance [2]. Ryder and Paull [18] present two incremental update algorithms, ACINCB and ACINCF, that allow incremental data-flow analysis. Yur et al. [26] propose an algorithm for an incremental *points-to* analysis. McPeak et al. [12] describe a technique for incremental and parallel static analysis based on *work units* (self-contained atoms of analysis input). The solutions are computed by a sort of processes called *analysis workers*, all coordinated by an *analysis master*. Also, there are papers that use memoization with a goal similar to the one of our cache, even if they

[3] Available at https://github.com/mcaos/incremental-mincaml

consider different analysis techniques. In particular, Mudduluru et al. propose, implement, and test an incremental analysis algorithm based on memoization of (equivalent) boolean formulas used to encode paths on programs [15]. Leino et al. [11] extend the verification machinery of Dafny with a way to cache the results from earlier runs of the verifier, so as to only verify those parts of the program most recently modified. Their cache mechanism is similar to ours, but no formal condition is explicitly stated guaranteeing a safe re-use of cached data, as we do. Also other authors apply memoization techniques to incremental model-checking [10,24] and incremental symbolic execution [25,17].

2 An overview of the incremental schema

In this section we illustrate how the algorithmic schema we propose can type check a simple program using incrementally the *standard* algorithm for non-interference by Volpano-Smith-Irvine [22,19] (see also Section 4). Suppose there is a large program P including the following code snippet that causes no leaks

$$x := y + z; \texttt{if } y + z \geq 42 \texttt{ then } result := y + z \texttt{ else } result := 42 \tag{1}$$

where $y, z, result$ are public and x is secret. After a shallow glance one may optimise it obtaining the following code that leaks the value of x

$$x := y + z; \texttt{if } x \geq 42 \texttt{ then } result := x \texttt{ else } result := 42 \tag{2}$$

Rather than re-typing the whole program P, one would like to detect the unsafe optimization by only type checking the *diff*, i.e. the `if-then-else`. To re-use as much as possible the typing information of P, we consider the abstract syntax tree of (1), we annotate its nodes with types, and we use this information to type (2). More precisely, we proceed as follows.

First, we build a *cache* C associating each statement with its type and the typing environment needed to obtain it. Then we *incrementally* use this information to decide which existing results in the cache can be re-used and which are to be recomputed for typing (2). This process is divided into four steps. For the moment, we omit the last one that consists in proving the correctness of the resulting algorithm, which is established by showing that a component of our construction (the predicate $compat_{env}$ used below) meets a mild condition.

Defining the shape of caches. The cache is a set of triples that associate with each statement the typing environment needed to close its free variables, and its type. For example, the first statement has the following entry in the cache, recording in the environment the types of the variables (H and L for secret and public) and the type $H\ cmd$ of the assignment (it is a command of type H),

$$(x := y + z, \{x \mapsto H\ var, y \mapsto L\ var, z \mapsto L\ var\}, H\ cmd) \tag{3}$$

Building caches. We visit the given annotated abstract syntax tree of (1) in a depth-first order and we cache the relevant triples for it and for its (sub-)trees. Consider again the first assignment for which the cache records the triple in (3), among others. All the entries for (1) are in Table 1.

Table 1: Tabular representation of the cache C for the program (1).

Expression	Environment	Type
(1)	$\{x \mapsto H\ var, y \mapsto L\ var, z \mapsto L\ var, result \mapsto L\ var\}$	$L\ cmd$
$x := y + z$	$\{x \mapsto H\ var, y \mapsto L\ var, z \mapsto L\ var\}$	$H\ cmd$
x	$\{x \mapsto H\ var\}$	H
y	$\{y \mapsto L\ var\}$	L
z	$\{z \mapsto L\ var\}$	L
$result$	$\{result \mapsto L\ var\}$	L
42	$\{\}$	L
$y + z$	$\{y \mapsto L\ var, z \mapsto L\ var\}$	L
$\texttt{if}\ y + z \geq 42\ \texttt{then}\ result := y + z\ \texttt{else}\ result := 42$	$\{y \mapsto L\ var, z \mapsto L\ var, result \mapsto L\ var\}$	$L\ cmd$
$y + z \geq 42$	$\{y \mapsto L\ var, z \mapsto L\ var\}$	L
$result := y + z$	$\{y \mapsto L\ var, z \mapsto L\ var, result \mapsto L\ var\}$	$L\ cmd$
$result := 42$	$\{result \mapsto L\ var\}$	$L\ cmd$

Incremental typing. The selected typing algorithm $\mathcal{S}$ is used to build the incremental algorithm $\mathcal{IS}$ as follows. A judgement inputs an environment Γ, a cache C and a statement c and it incrementally computes the type ς (see Section 4) and C', with possibly updated cache entries for the sub-terms of c:

$$\Gamma, C \vdash_{\mathcal{IS}} c : \varsigma \triangleright C'$$

The incremental algorithm is expressed as a set of inductively defined rules. Most of these simply mimic the structure of the rules defining $\mathcal{S}$. Consider the assignment that requires two rules. The first rule says that we can reuse the information available if the statement is cached and the environments Γ and Γ' coincide on the free variables of c (checked by the predicate $compat_{env}(\Gamma, \Gamma', c)$):

$$\frac{C(x := a) = \langle \Gamma', \varsigma \rangle \quad compat_{env}(\Gamma, \Gamma', x := a)}{\Gamma, C \vdash_{\mathcal{IS}} x := a : \varsigma \triangleright C}$$

The second rule is for when nothing is cached (the side condition *miss* holds), or the typing environments are not compatible. In this case, to obtain C', the new cache, is obtained from C by inserting in it the triples for x, for the expression a and for the assignment itself through $\mathcal{IS}$.

$$\frac{\begin{array}{l}\Gamma, C \vdash_{\mathcal{IS}} x : \tau_x \triangleright C'' \qquad \Gamma, C \vdash_{\mathcal{IS}} a : \tau_a \triangleright C''' \qquad \tau_a = \tau_x \\ \varsigma = \tau_a\ cmd \qquad C' = C'' \cup C''' \cup \{(x := a, \Gamma_{|\,FV(x:=a)}, \varsigma)\}\end{array}}{\Gamma, C \vdash_{\mathcal{IS}} x := a : \varsigma \triangleright C'}\ miss(C, x := a, \Gamma)$$

Back to our example, to discover the leak it suffices to type $x \geq 42$ and $result := x$ having already in the cache the types of x and *result*, while no re-typing is needed for all the other statements of the whole program P.

3 Formalizing the incremental schema

Here we formalise our algorithmic schema for incremental typing, exemplified in Section 2. Remarkably, it is independent of both the specific type system

and the programming language (for that we use below $t \in Term$ to denote an expression or a statement). We only assume to have variables $x, y, \ldots \in Var$, types $\tau, \tau', \ldots \in Type$, typing environments $\Gamma\colon Var \to Type \in Env$; and in addition that the original typing algorithm $\mathcal{A}$ is syntax-directed and defined through inference rules; that it is invoked by writing $\Gamma \vdash_{\mathcal{A}} t\colon R$, where $R \in Res$ is the result (not necessarily a type only).

Below we express the rules of $\mathcal{A}$ according to the following format. It is convenient to order the subterms of t, by stipulating $i \leq j$ provided that t_j requires the result of t_i to be typed ($i, j \leq n_t$).

$$\frac{\forall i \in \mathbb{I}_t \,.\, tr^t_{t_i}(\Gamma, \{R_j\}_{j<i \,\wedge\, j \in \mathbb{I}_t}) \vdash_{\mathcal{A}} t_i : R_i \qquad checkJoin_t(\Gamma, \{R_i\}_{i \in \mathbb{I}_t}, \mathtt{out}\ R)}{\Gamma \vdash_{\mathcal{A}} t : R}$$

where $\mathbb{I}_t \subseteq \{1, \ldots, n_t\}$. The function $tr^t_{t_i}$ maps Γ and a set of typing results R_i into the typing environment needed by t_i. The (conjunction of) predicate(s) $checkJoin_t$ checks that the subterms have compatible results R_i and combines them in the overall result R. (Both tr and $checkJoin$ are easily defined when typing rules in the usual format are rendered in the format above.)

For example the standard typing rule for variables:[4]

$$\frac{x \in dom(\Gamma) \qquad \tau = \Gamma(x)}{\Gamma \vdash_{\mathcal{A}} x : \tau}$$

is rendered in our format as follows (note that $\mathbb{I}_x = \emptyset$ just as the function tr)

$$\frac{checkJoin_x(\Gamma, \emptyset, \mathtt{out}\ \tau)}{\Gamma \vdash_{\mathcal{A}} x : \tau} \quad \text{where } checkJoin_x(\Gamma, \emptyset, \mathtt{out}\ \tau) \triangleq x \in dom(\Gamma) \wedge \tau = \Gamma(x)$$

As a further example consider the rule for the expression $\mathbf{let}\ x = e_2\ \mathbf{in}\ e_3$ below

$$\frac{\Gamma \vdash_{\mathcal{A}} e_2 : \tau_2 \qquad \Gamma[x \mapsto \tau_2] \vdash_{\mathcal{A}} e_3 : \tau_3}{\Gamma \vdash_{\mathcal{A}} \mathbf{let}\ x = e_2\ \mathbf{in}\ e_3 : \tau_3}$$

that becomes as follows (we abuse the set notation, e.g. omitting $\emptyset$ or { and }).

$$\frac{\begin{array}{c} tr^{\mathbf{let}\ x = e_2\ \mathbf{in}\ e_3}_{e_2}(\Gamma, \emptyset) \vdash_{\mathcal{A}} e_2 : \tau_2 \\ tr^{\mathbf{let}\ x = e_2\ \mathbf{in}\ e_3}_{e_3}(\Gamma, \tau_2) \vdash_{\mathcal{A}} e_3 : \tau_3 \qquad checkJoin_{\mathbf{let}\ x = e_2\ \mathbf{in}\ e_3}(\Gamma, \tau_2, \tau_3, \mathtt{out}\ \tau) \end{array}}{\Gamma \vdash_{\mathcal{A}} \mathbf{let}\ x = e_2\ \mathbf{in}\ e_3 : \tau}$$

Note that the definition of function tr is immediate; that we need the type of e_2 for typing e_3; and that the second parameter of $tr^{\mathbf{let}\ x = e_2\ \mathbf{in}\ e_3}_{e_2}$ is empty, because we only need the enviroment to type e_2.

$$tr^{\mathbf{let}\ x = e_2\ \mathbf{in}\ e_3}_{e_2}(\Gamma, \emptyset) \triangleq \Gamma \qquad\qquad tr^{\mathbf{let}\ x = e_2\ \mathbf{in}\ e_3}_{e_3}(\Gamma, \tau) \triangleq \Gamma[x \mapsto \tau] \qquad (4)$$

[4] Instead with the axiom $\Gamma'[x \mapsto \tau] \vdash_{\mathcal{A}} x : \tau$ one has $\mathbb{I}_x = \emptyset$ and the same $checkJoin_x$, where $\Gamma = \Gamma'[x \mapsto \tau]$.

Also the following definition is immediate

$$checkJoin_{\mathbf{let}\ x=e_2\ \mathbf{in}\ e_3}(\Gamma, \tau_2, \tau_3, \mathtt{out}\ \tau) \triangleq (\tau = \tau_3)$$

To enhance readability, we will hereto highlight the occurrences of $tr^t_{t'}$ (red in the pdf) and $checkJoin_t$ (blue in the pdf).

Defining the shape of caches. The shape of the cache is crucial for re-using incrementally portions of the available typing results. A cache associates the input data t and Γ with the result R, rendered by a set of triples (t, Γ, R), as done in Section 2. More formally, the set of caches C is defined as:

$$Cache = \wp(Terms \times Env \times Res)$$

We write $C(t) = \langle \Gamma, R \rangle$ if the cache has an entry for t, and $C(t) = \bot$ otherwise.

Building caches. Given a term, we assume that the nodes of its abstract syntax tree (called *annotated abstract syntax tree* or *aAST*) are annotated with the result of the typing for the subterm they represent (written $t : R$, possibly $t : \bot$ if t does not type). Let $\mathbb{I}_t$, $\{t_i\}_{i\in\mathbb{I}_t}$, and $tr^t_{t_i}$ be as above, and let $\Gamma_{|\,FV(t)}$ be the restriction of Γ to the free variables of t. Then the following procedure visits the aAST in a depth-first manner and builds the cache.

$$\begin{aligned} buildCache\ (t : R)\ \Gamma\ = & \{(t, \Gamma_{|\,FV(t)}, R)\} \cup \\ & \bigcup_{i\in\mathbb{I}_t} \left(buildCache\ (t_i : R_i)\ tr^t_{t_i}(\Gamma, \{R_j\}_{j<i\,\wedge\, j\in\mathbb{I}_t})\right) \end{aligned}$$

The following theorem ensures that each entry of a cache returned by *buildCache* represents correct typing information.

Theorem 1 (Cache correctness). *For all* t, R, Γ

$$(t, \Gamma, R) \in (buildCache\ (t : R)\ \Gamma) \iff \Gamma \vdash_{\mathcal{A}} t : R$$

Incremental typing. The third step consists of instantiating the rule templates that make typing incremental. We remark that no change to the original algorithm $\mathcal{A}$ is needed: it is used as a *grey-box* — what matters are just the shape of the original judgements, the rules and some domain-specific knowledge. The judgements for the incremental typing algorithm $\mathcal{IA}$ have the form:

$$\Gamma, C \vdash_{\mathcal{IA}} t : R \triangleright C'$$

We have three different rule templates defining the incremental typing algorithm.

The first template is for the case when there is a cache hit:

$$\frac{C(t) = \langle \Gamma', R \rangle \quad compat_{env}(\Gamma, \Gamma', t)}{\Gamma, C \vdash_{\mathcal{IA}} t : R \triangleright C}$$

where $compat_{env}(\Gamma, \Gamma', t)$ is a predicate testing the compatibility of typing environments for the term t and means that Γ' includes the information represented by Γ for t and that they are compatible (see the example in Section 2). Note that this predicate must be defined for *each* algorithm $\mathcal{A}$ and, as discussed below, it must meet a mild requirement to make the algorithm $\mathcal{IA}$ coherent with $\mathcal{A}$.

The second rule template is for when there is a cache miss and the term in hand has no subterms:

$$\frac{\Gamma \vdash_{\mathcal{A}} t : R \quad C' = C \cup \{(t, \Gamma_{|\,FV(t)}, R)\}}{\Gamma, C \vdash_{\mathcal{IA}} t : R \triangleright C'} \; miss(C, t, \Gamma)$$

where $\Gamma \vdash_{\mathcal{A}} t : R$ is the invocation to $\mathcal{A}$, and the predicate *miss* is defined below with the intuition that either there is no association for t in C, or if an association (t, Γ', R) exists the typing environment Γ' is not compatible with the current Γ.

$$miss(C, t, \Gamma) \triangleq \nexists \Gamma', R.\ \big(C(t) = \langle \Gamma', R \rangle \wedge compat_{env}(\Gamma, \Gamma', t)\big)$$

Finally, the last template applies when there is a cache miss, but the term t is inductively defined starting from its subterms. In this case the rule invokes the incremental algorithm on the subterms, by composing the results available in the cache (if any):

$$\frac{\begin{array}{c}\forall i \in \mathbb{I}_t\,.\, \boxed{tr_t^{t_i}(\Gamma, \{R_j\}_{j<i \wedge j \in \mathbb{I}_t})}, C \vdash_{\mathcal{IA}} t_i : R_i \triangleright C^i \\ \boxed{checkJoin_t(\Gamma, \{R_i\}_{i \in \mathbb{I}_t}, \texttt{out}\ R)} \quad C' = \{(t, \Gamma_{|\,FV(t)}, R)\} \cup \bigcup_{i \in \mathbb{I}_t} C^i\end{array}}{\Gamma, C \vdash_{\mathcal{IA}} t : R \triangleright C'} \; miss(C, t, \Gamma)$$

Typing coherence. The resulting algorithm $\mathcal{IA}$ preserves the correctness of the original one $\mathcal{A}$, provided that the rule templates above, and especially the predicate $compat_{env}$ are carefully instantiated.

The following definition characterises when two environments are compatible, and it helps in proving that our incremental typing correctly implements the given non-incremental one.

Definition 1 (Typing environment compatibility). *A predicate* $compat_{env}$ expresses compatibility *iff*

$$\forall\, \Gamma, \Gamma', t\,.\, compat_{env}(\Gamma, \Gamma', t) \wedge \Gamma' \vdash_{\mathcal{A}} t : R \implies \Gamma \vdash_{\mathcal{A}} t : R$$

Note that the notion of compatibility guarantees that Γ and Γ' share all the information needed to correctly type the term t. This is the basic condition to ensure that the incremental typing algorithm is concordant with the original one. In particular, the following theorem suffices to establish the correctness of the incremental algorithm $\mathcal{IA}$, provided that the original algorithm $\mathcal{A}$ is such. In its statement, the cache is universally quantified because $\mathcal{IA}$ re-uses $\mathcal{A}$ to re-build the needed cache as soon as a cache miss occurs.

Theorem 2 (Typing coherence). *If* $compat_{env}$ *expresses compatibility, then for all terms* t*, caches* C*, typing environments* Γ*, and typing algorithm* $\mathcal{A}$

$$\Gamma \vdash_{\mathcal{A}} t : R \iff \Gamma, C \vdash_{\mathcal{IA}} t : R \triangleright C'.$$

4 Incremental type checking for non-interference

Here we use incrementally the typing algorithm $\mathcal{S}$ of Volpano-Smith-Irvine [22,19] for checking non-interference policies, obtaining the algorithm $\mathcal{IS}$. We assume that the variables of programs are classified either as high, H, or low L. Intuitively, a program enjoys the non-interference property when the values of low level variables do not depend on those of high level ones.

As usual, assume a simple imperative language WHILE, whose syntax is below (Var denotes the set of program variables).

$$\begin{array}{ll}
AExpr \ni a ::= n \mid x \mid a_1\ \mathtt{op_a}\ a_2 & n \in \mathbb{N},\quad \mathtt{op_a} \in \{+, *, -, \ldots\},\quad x \in Var \\
BExpr \ni b ::= \mathtt{true} \mid \mathtt{false} \mid b_1\ \mathtt{or}\ b_2 \mid\ \mathtt{not}\ b \mid a_1 \leq a_2 & \\
Stmt \ni c ::= \mathtt{skip} \mid x := a \mid c_1; c_2 \mid \mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2 \mid \mathtt{while}\ b\ \mathtt{do}\ c & \\
Phrase \ni p ::= a \mid b \mid c & \\
DType \ni \tau ::= H \mid L \quad PType \ni \varsigma ::= \tau \mid \tau\ var \mid \tau\ cmd \quad Env \ni \Gamma ::= \emptyset \mid \Gamma[p \mapsto \varsigma] &
\end{array}$$

The type checking algorithm has judgements of the form

$$\Gamma \vdash_{\mathcal{S}} p : \varsigma$$

where $\varsigma \in PType = Res$, and its rules are in the extended version [3]. We have coloured and framed the results of tr and $\lceil checkJoin \rceil$.In the following we assume that the initial typing environment Γ contains the security level of each variable occurring in the program at hand.

Defining the shape of caches. The shape of the caches is:

$$C \in Cache = \wp(Phrase \times Env \times PType)$$

Building caches. We build the cache by visiting the aAST and "reconstructing" the typing environment. The function *buildCache* is in Fig. 1, where for brevity we have directly used the results of tr rather than writing the needed invocations.

Incremental typing. In Fig. 2 we display the rules defining the algorithm $\mathcal{IS}$ with judgements of the following form

$$\Gamma, C \vdash_{\mathcal{IS}} p : \varsigma \triangleright C'$$

Most of the rules are trivial instantiations of rules in Section 3 that mimic those of the original type checking algorithm. Of course, $\mathcal{IS}$ inherits unchanged the subtyping relation of $\mathcal{S}$ and applies it when needed.

Typing coherence. To prove that $\mathcal{IS}$ is coherent with $\mathcal{S}$, we first show that $compat_{env}$ satisfies Definition 1.

Lemma 1. *The predicate $compat_{env}$ of Eq. (5) in Fig. 2 expresses compatibility.*

The above lemma suffices to prove the following theorem, which is an instance of Theorem 2.

Theorem 3. $\forall \Gamma, C, e.\ \Gamma \vdash_{\mathcal{S}} e : \tau \iff \Gamma, C \vdash_{\mathcal{IS}} e : \tau \triangleright C'$

$$
\begin{aligned}
&\mathit{buildCache}\ (c : L)\ \Gamma \triangleq \{(c, \emptyset, L)\} \qquad c \in \mathbb{N} \cup \{\mathit{true}, \mathit{false}\}\\
&\mathit{buildCache}\ (x : \tau)\ \Gamma \triangleq \{(x, [x \mapsto \tau\ \mathit{var}], \tau)\}\\
&\mathit{buildCache}\ (a_1\ \texttt{op}\ a_2 : \tau)\ \Gamma \triangleq \{(a_1\ \texttt{op}\ a_2, \Gamma_{|\ FV(a_1 \texttt{op} a_2)}, \tau)\}\\
&\qquad \cup (\mathit{buildCache}\ (a_1 : \tau_1)\ \boxed{\Gamma}) \cup (\mathit{buildCache}\ (a_2 : \tau_2)\ \boxed{\Gamma})\\
&\mathit{buildCache}\ (a_1 \leq a_2 : \tau)\ \Gamma \triangleq \{(a_1 \leq a_2, \Gamma_{|\ FV(a_1 \leq a_2)}, \tau)\}\\
&\qquad \cup (\mathit{buildCache}\ (a_1 : \tau_1)\ \boxed{\Gamma}) \cup (\mathit{buildCache}\ (a_2 : \tau_2)\ \boxed{\Gamma})\\
&\mathit{buildCache}\ (b_1\ \texttt{or}\ b_2 : \tau)\ \Gamma \triangleq \{(b_1\ \texttt{or}\ b_2, \Gamma_{|\ FV(b_1 \texttt{or} b_2)}, \tau)\}\\
&\qquad \cup (\mathit{buildCache}\ (b_1 : \tau_1)\ \boxed{\Gamma}) \cup (\mathit{buildCache}\ (b_2 : \tau_2)\ \boxed{\Gamma})\\
&\mathit{buildCache}\ (\texttt{not}\ b : \tau)\ \Gamma \triangleq \{(\texttt{not}\ b, \Gamma_{|\ FV(\texttt{not} b)}, \tau)\} \cup (\mathit{buildCache}\ (b : \tau)\ \boxed{\Gamma})\\
&\mathit{buildCache}\ (\mathit{skip} : H\ \mathit{cmd})\ \Gamma \triangleq \{(\mathit{skip}, \emptyset, H\ \mathit{cmd})\}\\
&\mathit{buildCache}\ (x := a : \tau\ \mathit{cmd})\ \Gamma \triangleq \{(x := a, \Gamma_{|\ FV(x:=a)}, \tau\ \mathit{cmd})\}\\
&\qquad \cup (\mathit{buildCache}\ (x : \tau_x)\ \boxed{\Gamma}) \cup (\mathit{buildCache}\ (a : \tau_a)\ \boxed{\Gamma})\\
&\mathit{buildCache}\ (\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 : \tau\ \mathit{cmd})\ \Gamma \triangleq \{(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \Gamma_{|\ FV(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2)}, \tau\ \mathit{cmd})\}\\
&\qquad \cup (\mathit{buildCache}\ (b : \tau_b)\ \boxed{\Gamma}) \cup (\mathit{buildCache}\ (c_1 : \tau_1\ \mathit{cmd})\ \boxed{\Gamma}) \cup (\mathit{buildCache}\ (c_2 : \tau_2\ \mathit{cmd})\ \boxed{\Gamma})\\
&\mathit{buildCache}\ (\texttt{while}\ b\ \texttt{do}\ c : \tau\ \mathit{cmd})\ \Gamma \triangleq \{(\texttt{while}\ b\ \texttt{do}\ c, \Gamma_{|\ FV(\texttt{while}\ b\ \texttt{do}\ c)}, \tau\ \mathit{cmd})\}\\
&\qquad \cup (\mathit{buildCache}\ (b : \tau_b)\ \boxed{\Gamma}) \cup (\mathit{buildCache}\ (c : \tau_c\ \mathit{cmd})\ \boxed{\Gamma})\\
&\mathit{buildCache}\ (c_1; c_2 : \tau\ \mathit{cmd})\ \Gamma \triangleq \{(c_1; c_2, \Gamma_{|\ FV(c_1; c_2)}, \tau\ \mathit{cmd})\}\\
&\qquad \cup (\mathit{buildCache}\ (c_1 : \tau_1\ \mathit{cmd})\ \boxed{\Gamma}) \cup (\mathit{buildCache}\ (c_2 : \tau_2\ \mathit{cmd})\ \boxed{\Gamma})
\end{aligned}
$$

Fig. 1: Definition of *buildCache* for the incremental type checking of WHILE.

5 Incremental type inference for a functional language

In this section we instantiate our schema in order to use incrementally the type inference algorithm of a simple functional programming language, called FUN. The syntax, the types and the semantics of FUN are standard, see e.g. [16]. Types are also now augmented with type variables $\alpha, \beta, \ldots \in \mathit{TVar}$. We only recall some relevant aspects below.

$$
\begin{aligned}
\mathit{Val} \ni v &::= c \mid \lambda_f\, x.e \qquad\qquad \texttt{op} \in \{+, *, =, \leq\}\\
\mathit{Expr} \ni e &::= v \mid x \mid e_1\ \texttt{op}\ e_2 \mid e_1\, e_2 \mid \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \mid \textbf{let}\ x = e_2\ \textbf{in}\ e_3\\
\mathit{AType} \ni \tau &::= \texttt{int} \mid \texttt{bool} \mid \tau_1 \to \tau_2 \mid \alpha \qquad \mathit{Env} \ni \Gamma ::= \emptyset \mid \Gamma[x \mapsto \tau]
\end{aligned}
$$

where in the functional abstraction f denotes the name of the (possibly) recursive function we are defining. The judgements of the type inference algorithm $\mathcal{W}$ are

$$\Gamma \vdash_{\mathcal{W}} e \colon (\tau, \theta)$$

where $\theta \colon (\mathit{TVar} \to \mathit{AType}) \in \mathit{Subst}$ is a substitution mapping type variables into augmented types. As usual, we write $\theta\,\tau$ to indicate the application of the substitution θ to τ, and $\theta_2 \circ \theta_1$ stands for the composition of substitutions.

Hereafter, we assume to use the inference algorithm $\mathcal{W}$ (see e.g. [16]), where constants c have a fixed and known type, and $\mathcal{U}$ denotes the standard type unification algorithm.

$$\frac{C(p) = \langle \Gamma', \varsigma \rangle \quad compat_{env}(\Gamma, \Gamma', p)}{\Gamma, C \vdash_{\mathcal{IS}} p : \varsigma \triangleright C} \ (\mathcal{IS}\text{-Hit}) \qquad \frac{\emptyset \vdash_{\mathcal{S}} c : \varsigma \quad C' = C \cup \{(c, \emptyset, \varsigma)\}}{\Gamma, C \vdash_{\mathcal{IS}} c : \varsigma \triangleright C'} \ miss(C, c, \Gamma) \ (\mathcal{IS}\text{-Const-Miss})$$

$$\frac{\Gamma \vdash_{\mathcal{S}} x : \varsigma \quad C' = C \cup \{(x, \Gamma_{|\,x}, \varsigma)\}}{\Gamma, C \vdash_{\mathcal{IS}} x : \varsigma \triangleright C'} \ miss(C, x, \Gamma) \ (\mathcal{IS}\text{-Var-Miss}) \qquad \frac{\Gamma \vdash_{\mathcal{S}} \texttt{skip} : \varsigma \quad C' = C \cup \{(\texttt{skip}, \emptyset, \varsigma)\}}{\Gamma, C \vdash_{\mathcal{IS}} \texttt{skip} : \varsigma \triangleright C'} \ miss(C, \texttt{skip}, \Gamma) \ (\mathcal{IS}\text{-Skip-Miss})$$

$$\frac{\boxed{\Gamma}, C \vdash_{\mathcal{IS}} a_1 : \tau_1 \triangleright C'' \quad \boxed{\Gamma}, C \vdash_{\mathcal{IS}} a_2 : \tau_2 \triangleright C''' \quad \boxed{\tau_1 = \tau_2 \wedge \varsigma = \tau_1} \quad C' = C'' \cup C''' \cup \{(a_1 \ \texttt{op} \ a_2, \Gamma_{|\,FV(a_1 \texttt{op} a_2)}, \varsigma)\}}{\Gamma, C \vdash_{\mathcal{IS}} a_1 \ \texttt{op} \ a_2 : \varsigma \triangleright C'} \ miss(C, a_1 \ \texttt{op} \ a_2, \Gamma) \ (\mathcal{IS}\text{-Op-Miss})$$

$$\frac{\boxed{\Gamma}, C \vdash_{\mathcal{IS}} b_1 : \tau_1 \triangleright C'' \quad \boxed{\Gamma}, C \vdash_{\mathcal{IS}} b_2 : \tau_2 \triangleright C''' \quad \boxed{\tau_1 = \tau_2 \wedge \varsigma = \tau_1} \quad C' = C'' \cup C''' \cup \{(b_1 \ \texttt{or} \ b_2, \Gamma_{|\,FV(b_1 \texttt{or} b_2)}, \varsigma)\}}{\Gamma, C \vdash_{\mathcal{IS}} b_1 \ \texttt{or} \ b_2 : \varsigma \triangleright C'} \ miss(C, b_1 \ \texttt{or} \ b_2, \Gamma) \ (\mathcal{IS}\text{-BOp-Miss})$$

$$\frac{\boxed{\Gamma}, C \vdash_{\mathcal{IS}} b : \tau \triangleright C'' \quad C' = C'' \cup \{(\ \texttt{not} \ b, \Gamma_{|\,FV(\texttt{not} b)}, \tau)\}}{\boxed{\Gamma}, C \vdash_{\mathcal{IS}} \ \texttt{not} \ b : \tau \triangleright C'} \ miss(C, \ \texttt{not} \ b, \Gamma) \ (\mathcal{IS}\text{-Not-Miss})$$

$$\frac{\boxed{\Gamma}, C \vdash_{\mathcal{IS}} a_1 : \tau_1 \triangleright C'' \quad \boxed{\Gamma}, C \vdash_{\mathcal{IS}} a_2 : \tau_2 \triangleright C''' \quad \boxed{\tau_1 = \tau_2 \wedge \varsigma = \tau_1} \quad C' = C'' \cup C''' \cup \{(a_1 \leq a_2, \Gamma_{|\,FV(a_1 \leq a_2)}, \varsigma)\}}{\Gamma, C \vdash_{\mathcal{IS}} a_1 \leq a_2 : \varsigma \triangleright C'} \ miss(C, a_1 \leq a_2, \Gamma) \ (\mathcal{IS}\text{-Leq-Miss})$$

$$\frac{\boxed{\Gamma}, C \vdash_{\mathcal{IS}} x : \tau_x \ var \triangleright C'' \quad \boxed{\Gamma}, C \vdash_{\mathcal{IS}} a : \tau_a \triangleright C''' \quad \boxed{\tau_a = \tau_x \wedge \varsigma = \tau_a \ cmd} \quad C' = C'' \cup C''' \cup \{(x := a, \Gamma_{|\,FV(x:=a)}, \varsigma)\}}{\Gamma, C \vdash_{\mathcal{IS}} x := a : \varsigma \triangleright C'} \ miss(C, x := a, \Gamma) \ (\mathcal{IS}\text{-Assign-Miss})$$

$$\frac{\begin{array}{c}\boxed{\Gamma}, C \vdash_{\mathcal{IS}} b : \tau_b \triangleright C'' \\ \boxed{\Gamma}, C \vdash_{\mathcal{IS}} c_1 : \tau_1 \ cmd \triangleright C''' \quad \boxed{\Gamma}, C \vdash_{\mathcal{IS}} c_2 : \tau_2 \ cmd \triangleright C^{iv} \quad \boxed{\tau_1 = \tau_2 = \tau_b \wedge \varsigma = \tau_1 \ cmd} \\ C' = C'' \cup C''' \cup C^{iv} \cup \{(\texttt{if} \ b \ \texttt{then} \ c_1 \ \texttt{else} \ c_2, \Gamma_{|\,FV(\texttt{if} \ b \ \texttt{then} \ c_1 \ \texttt{else} \ c_2)}, \varsigma)\}\end{array}}{\Gamma, C \vdash_{\mathcal{IS}} \texttt{if} \ b \ \texttt{then} \ c_1 \ \texttt{else} \ c_2 : \varsigma \triangleright C'} \ miss(C, \texttt{if} \ b \ \texttt{then} \ c_1 \ \texttt{else} \ c_2, \Gamma) \ (\mathcal{IS}\text{-If-Miss})$$

$$\frac{\begin{array}{c}\boxed{\Gamma}, C \vdash_{\mathcal{IS}} b : \tau_b \triangleright C'' \\ \boxed{\Gamma}, C \vdash_{\mathcal{IS}} c : \tau_1 \ cmd \triangleright C''' \quad \boxed{\tau_1 = \tau_b \wedge \varsigma = \tau_1 \ cmd} \\ C' = C'' \cup C''' \cup \{(\texttt{while} \ b \ \texttt{do} \ c, \Gamma_{|\,FV(\texttt{while} \ b \ \texttt{do} \ c)}, \varsigma)\}\end{array}}{\Gamma, C \vdash_{\mathcal{IS}} \texttt{while} \ b \ \texttt{do} \ S : \varsigma \triangleright C'} \ miss(C, \texttt{while} \ b \ \texttt{do} \ c, \Gamma) \ (\mathcal{IS}\text{-While-Miss})$$

$$\frac{\boxed{\Gamma}, C \vdash_{\mathcal{IS}} c_1 : \tau_1 \ cmd \triangleright C'' \quad \boxed{\Gamma}, C \vdash_{\mathcal{IS}} c_2 : \tau_2 \ cmd \triangleright C''' \quad \boxed{\tau_1 = \tau_2 \wedge \varsigma = \tau_1 \ cmd} \quad C' = C'' \cup C''' \cup \{(c_1; c_2, \Gamma_{|\,FV(c_1; c_2)}, \varsigma)\}}{\Gamma, C \vdash_{\mathcal{IS}} c_1; c_2 : \varsigma \triangleright C'} \ miss(C, c_1; c_2, \Gamma) \ (\mathcal{IS}\text{-Seq-Miss})$$

$$\text{with } compat_{env}(\Gamma, \Gamma', p) \triangleq dom(\Gamma) \supseteq FV(p) \wedge dom(\Gamma') \supseteq FV(p) \wedge \forall y \in FV(p) \,.\, \Gamma(y) = \Gamma'(y) \quad (5)$$

Fig. 2: Rules defining the incremental algorithm $\mathcal{IS}$ to type check WHILE.

$$
\begin{aligned}
&\mathit{buildCache}\ (c : (\tau_c, \theta))\ \Gamma \triangleq \{(c, \emptyset, (\tau_c, \theta))\}\\
&\mathit{buildCache}\ (x : (\tau_x, \theta))\ \Gamma \triangleq \{(x, [x \mapsto \tau_x], (\tau_x, \theta))\}\\
&\mathit{buildCache}\ (\lambda_f\, x.e : (\tau_f, \theta_f))\ \Gamma \triangleq \{(\lambda_f\, x.e, \Gamma_{|\, FV(\lambda_f\, x.e)}, (\tau_f, \theta_f))\} \cup (\mathit{buildCache}\ (f : (\tau_f, \theta_f))\ \boxed{\Gamma})\\
&\qquad \cup (\mathit{buildCache}\ (x : (\tau_x, \theta_x))\ \boxed{\Gamma}) \cup (\mathit{buildCache}\ (e : (\tau_e, \theta_e))\ \boxed{\Gamma[x \mapsto \tau_x, f \mapsto \tau_f]})\\
&\mathit{buildCache}\ (\mathbf{let}\ x = e_2\ \mathbf{in}\ e_3 : (\tau_{let}, \theta_{let}))\ \Gamma \triangleq \{(\mathbf{let}\ x = e_2\ \mathbf{in}\ e_3, \Gamma_{|\, FV(\mathbf{let}\ x = e_2\ \mathbf{in}\ e_3)}, (\tau_{let}, \theta_{let}))\}\\
&\qquad \cup (\mathit{buildCache}\ (x : (\tau_x, \theta_x))\ \boxed{\Gamma}) \cup (\mathit{buildCache}\ (e_2 : (\tau_2, \theta_2))\ \boxed{\Gamma})\\
&\qquad \cup (\mathit{buildCache}\ (e_3 : (\tau_3, \theta_3))\ \boxed{\Gamma[x \mapsto \tau_x]})\\
&\mathit{buildCache}\ (e_1\ \mathtt{op}\ e_2 : (\tau_{op}, \theta_{op}))\ \Gamma \triangleq \{(e_1\ \mathtt{op}\ e_2, \Gamma_{|\, FV(e_1 \mathtt{op} e_2)}, (\tau_{op}, \theta_{op}))\}\\
&\qquad \cup (\mathit{buildCache}\ (e_1 : (\tau_1, \theta_1)\ \boxed{\Gamma}) \cup (\mathit{buildCache}\ (e_2 : (\tau_2, \theta_2)\ \boxed{\Gamma})\\
&\mathit{buildCache}\ (e_1\, e_2 : (\tau_{app}, \theta_{app}))\ \Gamma \triangleq \{(e_1\, e_2, \Gamma_{|\, FV(e_1\, e_2)}, (\tau_{app}, \theta_{app}))\}\\
&\qquad \cup (\mathit{buildCache}\ (e_1 : (\tau_1, \theta_1))\ \boxed{\Gamma}) \cup (\mathit{buildCache}\ (e_2 : (\tau_2, \theta_2))\ \boxed{\Gamma})
\end{aligned}
$$

Fig. 3: Definition of *buildCache* for the incremental type inference of FUN.

Defining the shape of caches. Entries in the cache are $(e, \Gamma, (\tau, \theta))$ and a cache is

$$C \in Cache = \wp(Expr \times Env \times (AType \times Subst))$$

Building caches. The function *buildCache* is easily defined in Fig. 3.

Incremental typing. In Fig. 4 we display the rules defining the algorithm $\mathcal{IW}$ with judgements of the following form

$$\Gamma, C \vdash_{\mathcal{IW}} e : (\tau, \theta) \triangleright C'$$

Most of the rules mimic the behaviour of algorithm $\mathcal{W}$, following the templates of Section 3. Consider for example the rule ($\mathcal{IW}$-LET-MISS): first, the types of e_1 and e_2 are incrementally inferred in the environments prescribed by the relevant calls to the function tr. The result associated with the whole expression **let-in** is then the pair $(\tau_2, \theta_2 \circ \theta_1)$, where θ_1 and θ_2 are the substitutions obtained recursively from e_1 and e_2, respectively.

Typing coherence. To prove the incremental algorithm $\mathcal{IW}$ coherent with $\mathcal{W}$, we first show that $compat_{env}$ satisfies Definition 1.

Lemma 2. *The predicate $compat_{env}$ of Eq. (6) in Fig. 4 expresses compatibility.*

Again, the following theorem is an instance of Theorem 2, and follows from the above lemma.

Theorem 4. $\forall \Gamma, C, e.\ \Gamma \vdash_{\mathcal{W}} e : (\tau, \theta) \iff \Gamma, C \vdash_{\mathcal{IW}} e : (\tau, \theta) \triangleright C'$

6 Implementation and some experiments

We have implemented in OCaml our proposal making incremental the usage the type-checker of MinCaml [20].[5] A formalization of MinCaml in our framework

[5] Available at https://github.com/mcaos/incremental-mincaml

$$\frac{C(e) = \langle \Gamma', (\tau, \theta)\rangle \quad compat_{env}(\Gamma, \Gamma', e)}{\Gamma, C \vdash_{\mathcal{IW}} e : (\tau, \theta) \triangleright C} \ (\mathcal{IW}\text{-Hit}) \qquad \frac{\Gamma \vdash_{\mathcal{W}} c : (\tau, \theta) \quad C' = C \cup \{(c, \emptyset, (\tau, \theta))\}}{\Gamma, C \vdash_{\mathcal{IW}} c : (\tau, \theta) \triangleright C'} \ (\mathcal{IW}\text{-Const-Miss}) \ miss(C, c, \Gamma)$$

$$\frac{\Gamma \vdash_{\mathcal{W}} x : (\tau, \theta) \quad C' = C \cup \{(x, \Gamma_{|\,x}, (\tau, \theta))\}}{\Gamma, C \vdash_{\mathcal{IW}} x : (\tau, \theta) \triangleright C'} \ (\mathcal{IW}\text{-Var-Miss}) \ miss(C, x, \Gamma)$$

$$\frac{\begin{array}{c}\boxed{\Gamma[x \mapsto \alpha_x, f \mapsto \alpha_x \to \alpha_e]}, C \vdash_{\mathcal{IW}} e : (\tau_e, \theta_e) \triangleright C'' \\ \theta_1 = \mathcal{U}(\tau_e, \theta_e \alpha_e) \wedge (\tau, \theta) = ((\theta_1\ (\theta_e\ \alpha_x)) \to (\theta_1\ \tau_e), \theta_1 \circ \theta_e) \\ C' = C'' \cup \{(\lambda_f\ x : \tau_x.e, \Gamma_{|\ FV(\lambda_f\ x:\tau_x.e}, (\tau, \theta))\}\end{array}}{\Gamma, C \vdash_{\mathcal{IW}} \lambda_f\ x.e : (\tau, \theta) \triangleright C'} \ (\mathcal{IW}\text{-Abs-Miss}) \ miss(C, \lambda_f\ x.e, \Gamma) \wedge \alpha_x, \alpha_e \ fresh$$

$$\frac{\begin{array}{c}\boxed{\Gamma}, C \vdash_{\mathcal{IW}} e_1 : (\tau_1, \theta_1) \triangleright C'' \quad \boxed{\theta_1\ \Gamma}, C \vdash_{\mathcal{IW}} e_2 : (\tau_2, \theta_2) \triangleright C''' \\ \theta_3 = \mathcal{U}(\theta_2\ \tau_1, \tau_{op}) \wedge \theta_4 = \mathcal{U}(\theta_3\ \tau_2, \tau_{op}) \wedge (\tau, \theta) = (\tau_{res}, \theta_4 \circ \theta_3 \circ \theta_2 \circ \theta_1) \\ C' = C'' \cup C''' \cup \{(e_1\ \texttt{op}\ e_2, \Gamma_{|\ FV(e_1 \texttt{op} e_2)}, (\tau, \theta))\}\end{array}}{\Gamma, C \vdash_{\mathcal{IW}} e_1\ \texttt{op}\ e_2 : (\tau, \theta) \triangleright C'} \ (\mathcal{IW}\text{-Op-Miss}) \ miss(C, e_1\ \texttt{op}\ e_2, \Gamma) \wedge \tau_{op}, \tau_{res} = \{\texttt{int}, \texttt{bool}\}$$

$$\frac{\begin{array}{c}\boxed{\Gamma}, C \vdash_{\mathcal{IW}} e_1 : (\tau_1, \theta_1) \to \tau_e \triangleright C'' \quad \boxed{\theta_1\ \Gamma}, C \vdash_{\mathcal{IW}} e_2 : (\tau_2, \theta_2) \triangleright C''' \\ \theta_3 = \mathcal{U}(\theta_2\ \tau_1, \tau_2 \to \alpha) \wedge (\tau, \theta) = (\theta_3\ \alpha, \theta_3 \circ \theta_2 \circ \theta_1) \\ C' = C'' \cup C''' \cup \{(e_1\ e_2, \Gamma_{|\ FV(e_1\ e_2)}, (\tau, \theta))\}\end{array}}{\Gamma, C \vdash_{\mathcal{IW}} e_1\ e_2 : (\tau, \theta) \triangleright C'} \ (\mathcal{IW}\text{-App-Miss}) \ miss(C, e_1\ e_2, \Gamma) \wedge \alpha \ \ fresh$$

$$\frac{\begin{array}{c}\boxed{\Gamma}, C \vdash_{\mathcal{IW}} e_1 : (\tau_1, \theta_1) \triangleright C'' \\ \boxed{\theta_1\ \Gamma}, C \vdash_{\mathcal{IW}} e_2 : (\tau_2, \theta_2) \triangleright C''' \quad \boxed{\theta_2(\theta_1\ \Gamma)}, C \vdash_{\mathcal{IW}} e_3 : (\tau_3, \theta_3) \triangleright C^{iv} \\ \theta_4 = \mathcal{U}(\theta_3(\theta_2\ \tau_1), bool) \wedge \theta_5 = \mathcal{U}(\theta_4\ \tau_3, \theta_4(\theta_3\ \tau_1)) \wedge (\tau, \theta) = (\theta_5(\theta_4\ \tau_3), \theta_5 \circ \theta_4 \circ \theta_3 \circ \theta_2) \\ C' = C'' \cup C''' \cup C^{iv} \cup \{(\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3, \Gamma_{|\ FV(\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3)}, (\tau, \theta))\}\end{array}}{\Gamma, C \vdash_{\mathcal{IW}} \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : (\tau, \theta) \triangleright C'} \ (\mathcal{IW}\text{-If-Miss}) \ miss(C, \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3, \Gamma)$$

$$\frac{\begin{array}{c}\boxed{\Gamma}, C \vdash_{\mathcal{IW}} e_2 : (\tau_2, \theta_2) \triangleright C'' \quad \boxed{(\theta_1\ \Gamma)[x \mapsto \tau_2]}, C \vdash_{\mathcal{IW}} e_3 : (\tau_3, \theta_3) \triangleright C''' \\ (\tau, \theta) = (\tau_3, \theta_3 \circ \theta_1) \quad C' = C'' \cup C''' \cup \{(\textbf{let}\ x = e_2\ \textbf{in}\ e_3, \Gamma_{|\ FV(\textbf{let}\ x = e_2\ \textbf{in}\ e_3)}, (\tau, \theta))\}\end{array}}{\Gamma, C \vdash_{\mathcal{IW}} \textbf{let}\ x = e_2\ \textbf{in}\ e_3 : (\tau, \theta) \triangleright C'} \ (\mathcal{IW}\text{-Let-Miss}) \ miss(C, \textbf{let}\ x = e_1\ \textbf{in}\ e_3, \Gamma)$$

$$\text{with } compat_{env}(\Gamma, \Gamma', e) \triangleq dom(\Gamma) \supseteq FV(e) \wedge dom(\Gamma') \supseteq FV(e) \wedge \forall y \in FV(e)\,.\,\mathcal{U}(\Gamma(y), \Gamma'(y)) \quad (6)$$

Fig. 4: Rules defining incremental algorithm $\mathcal{IW}$ to infer FUN types.

and all tables and results of our experiments are available online in the extended version [3]. In detail, caches and type environments are implemented as hash-tables, so their handling is done almost in constant time. The memory overhead due to the cache is $\mathcal{O}(n \times m)$, where n is the size of the program under analysis and m is the number of variables therein.

The other possible time consuming part concerns checking environment compatibility. The key idea to make $compat_{env}$ efficient is to compute the sets of the free variables beforehand, and to store them as additional annotations on

the aAST. Summing up, implementing our schema is not too demanding, since it can be done with standard data structures.

Next, we show that (i) the cost of using the type checker incrementally depends on the size of *diffs*; (ii) its performance increases as these become smaller; and (iii) the incremental usage is almost always faster than re-using the standard one. The comparison is done by type checking synthetic programs with (binary and complete) aAST of increasing depth from 8 to 16, and with a number of variables ranging from 1 to 2^{15}. All the internal nodes are binary operators and the leaves are free variables. This test suites are intended to stress our incremental algorithm in the worst, yet artificial case. The measures are obtained using the library *Benchmark* that takes into account the overhead of OCaml runtime.[6]

To test the efficiency of caching we first re-typed twice the program with no changes, starting with an empty cache. We considered (binary and complete) aAST of depth 16, and we collected the number of re-typings per second in function of the number of variables in the program, ranging from 1 to 2^{15}.

The experiments show not only that the overhead for caching is negligible but also that caching is beneficial when the number of free variables is not too large w.r.t. the aAST depth because the results of common subtrees are re-used.

Then, we have simulated program changes by invalidating parts of caches that correspond to the rightmost subexpression at different depths. Note that invalidating cache entries for the *diff* subexpression e' of e requires to invalidate (i) all the entries for the nodes in the path from the root of the aAST of e to e' and (ii) all the entries for e' and its subexpressions, recursively. We collected the number of re-typings per second vs. the size of the *diff* for a few choices of aAST depth (from 12 tp 16) and number of variables (from 2^7 to 2^{16}). The experimental results show that our caching and memoization is faster than re-typing twice. An exception is when aAST have the maximum number of variables and the considered changes exceed 25% of the nodes. All in all, the advantage of using incrementally a type checker decreases, as expected, when there is a significant growth of the number of variables or in the size of the program. However, these cases only show up with very big numbers, which are not likely to occur often.

We also measured the number of MBs allocated by our implementation for the standard type checking and by its incremental usage with respect to the size of the synthetic aAST and the number of free variables.[7] We considered aAST of depth ranging from 10 to 16 and the number of variables ranging from 2^9 and 2^{15}, and the results show that the ratio standard/incremental is .99, almost constant.

7 Conclusions

We have presented an algorithmic schema for incrementally using existing type checking and type inference algorithms. Since only the shape of the input, the output, and some domain-specific knowledge of the original algorithms are relevant, our schema considers them as grey-boxes. Remarkably, the only real

[6] Available at https://github.com/Chris00/ocaml-benchmark

[7] As measured by the Landmarks library https://github.com/LexiFi/landmarks

effort for defining the incremental algorithm is required for establishing the notion of compatibility between parts of the environments relevant for re-typing. We have introduced the basic bricks of our approach and proved a theorem guaranteeing the coherence of *any* original algorithm with its incremental version, and *vice versa*. As a matter of fact, coherence follows from easily checking a mild condition on the environment compatibility. To illustrate the approach we have then instantiated our proposal for checking non-interference within an imperative language and for type inference within a functional language.

We have implemented the incremental version of the type checker of MinCaml, and we have assessed it on synthetic programs with varying size and number of variables. The experiments have shown our proposal worth using within a continuous software development model where fast responsiveness is needed, because only *diffs* are typed, possibly with those parts of the code affected by them. Additionally, the cost of using the type checker incrementally depends on the size of *diffs*, and its performance increases as these become smaller, a typical situation when applying local transformations, e.g. code motion, dead code elimination, and code wrapping.

Future work. We are confident that little extensions to our proposal are needed to cover also type and effect systems. Also, other programming paradigms should be easily accommodated in our incremental schema, as preliminary results on process calculi suggest. More work is instead required to apply our ideas to other syntax-directed static analyses, e.g. control flow analysis because of fixed-point computations. We also plan to carry our proposal on Abstract Interpretation, where the rich structure of the abstract domains poses some serious challenges. Presently, we are extending our prototype with an incremental type inference for MinCaml. Moreover, we plan to further automatize our proposal by mechanically deducing the relation $compat_{env}$, based on the syntax and on some relevant aspects of types, e.g. sub-typing. Since *tr* and *checkJoin* are directly inherited from the given type checking or inference algorithm, one can implement a generator that automatically produces its corresponding incremental version.

More experiments on real programs are also in order to better assess the performance of our proposal, as well as its scalability.

Finally, we would also like to apply the incremental schema to real-world languages, e.g. OCaml.

References

1. Aditya, S., Nikhil, R.S.: Incremental polymorphism. In: Hughes, J. (ed.) Functional Programming Languages and Computer Architecture, 5th ACM, Proceedings. LNCS, vol. 523, pp. 379–405. Springer (1991)
2. Blackshear, S., Di Stefano, D., Luca, M., O'Hearn, P., Villard, J.: Finding inter-procedural bugs at scale with infer static analyzer (Sep 2017), https://code.facebook.com/posts/1537144479682247/finding-inter-procedural-bugs-at-scale-with-infer-static-analyzer/

3. Busi, M., Degano, P., Galletta, L.: Using standard typing algorithms incrementally, extended version available at abs/1808.00225
4. Calcagno, C., Di Stefano, D., Dubreil, J., Gabi, D., Hooimeijer, P., Luca, M., O'Hearn, P.W., Papakonstantinou, I., Purbrick, J., Rodriguez, D.: Moving fast with software verification. In: Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods. pp. 3–11 (2015)
5. Erdweg, S., Bracevac, O., Kuci, E., Krebs, M., Mezini, M.: 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. pp. 880–897 (2015)
6. Facebook: Pyre - a performant type-checker for Python 3, https://pyre-check.org/
7. Harman, M., O'Hearn, P.: From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis. In: IEEE International Working Conference on Source Code Analysis and Manipulation (2018)
8. Infer, F.: Infer static analyzer, http://fbinfer.com/
9. Johnson, G.F., Walz, J.A.: A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In: Proceedings of the 13th Symposium on Principles of Programming Languages. pp. 44–57. ACM (1986)
10. Lauterburg, S., Sobeih, A., Marinov, D., Viswanathan, M.: Incremental state-space exploration for programs with dynamically allocated data. In: 30th International Conference on Software Engineering (ICSE 2008). pp. 291–300 (2008)
11. Leino, K.R.M., Wüstholz, V.: Fine-grained caching of verification results. In: Kroening, D., Pasareanu, C.S. (eds.) Computer Aided Verification. LNCS, vol. 9206, pp. 380–397. Springer (2015)
12. McPeak, S., Gros, C., Ramanathan, M.K.: 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. pp. 554–564 (2013)
13. Meertens, L.G.L.T.: Incremental polymorphic type checking in B. In: Wright, J.R., Landweber, L., Demers, A.J., Teitelbaum, T. (eds.) Proceedings of the 10th ACM Symposium on Principles of Programming Languages. pp. 265–275. ACM (1983)
14. Miao, W., Siek, J.G.: Incremental type-checking for type-reflective metaprograms. In: Visser, E., Järvi, J. (eds.) Proceedings of the Ninth International Conference on Generative Programming and Component Engineering. pp. 167–176. ACM (2010)
15. Mudduluru, R., Ramanathan, M.K.: Efficient incremental static analysis using path abstraction. In: Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014. pp. 125–139 (2014)
16. Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. Springer (1999)
17. Qiu, R., Yang, G., Pasareanu, C.S., Khurshid, S.: Compositional symbolic execution with memoized replay. In: 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Volume 1. pp. 632–642 (2015)
18. Ryder, B.G., Paull, M.C.: Incremental data-flow analysis. ACM Trans. Program. Lang. Syst. **10**(1), 1–50 (1988)
19. Smith, G.: Principles of secure information flow analysis. In: Malware Detection, pp. 291–307. Springer (2007)
20. Sumii, E.: Mincaml: a simple and efficient compiler for a minimal functional language. In: Findler, R.B., Hanus, M., Thompson, S. (eds.) Proceedings of the 2005 workshop on Functional and declarative programming in education. pp. 27–38. ACM (2005)

21. Szabó, T., Erdweg, S., Voelter, M.: Inca: a DSL for the definition of incremental program analyses. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. pp. 320–331 (2016)
22. Volpano, D.M., Irvine, C.E., Smith, G.: A sound type system for secure flow analysis. Journal of Computer Security **4**(2/3), 167–188 (1996)
23. Wachsmuth, G., Konat, G.D.P., Vergu, V.A., Groenewegen, D.M., Visser, E.: A language independent task engine for incremental name and type analysis. In: Erwig, M., Paige, R.F., Wyk, E.V. (eds.) Software Language Engineering - 6th International Conference. LNCS, vol. 8225, pp. 260–280. Springer (2013)
24. Yang, G., Dwyer, M.B., Rothermel, G.: Regression model checking. In: 25th IEEE International Conference on Software Maintenance (ICSM 2009). pp. 115–124 (2009)
25. Yang, G., Person, S., Rungta, N., Khurshid, S.: Directed incremental symbolic execution. ACM Trans. Softw. Eng. Methodol. **24**(1), 3:1–3:42 (2014)
26. Yur, J., Ryder, B.G., Landi, W.: An incremental flow- and context-sensitive pointer aliasing analysis. In: Proceedings of the 1999 International Conference on Software Engineering. pp. 442–451 (1999)