Safe Incremental Type Checking

Matthias Puech, Yann Régis-Gianas

Safe Incremental Type Checking

Matthias Puech
Department of Comp. Sci., Univ. of Bologna,
PPS, Team πr^2 (Univ. Paris Diderot, CNRS, INRIA)
puech@cs.unibo.it

Yann Régis-Gianas
PPS, Team πr^2 (Univ. Paris Diderot, CNRS, INRIA)
yrg@pps.jussieu.fr

Abstract

We study the problem of verifying the well-typing of terms, not in a batch fashion, as it is usually the case for typed languages, but incrementally, that is by sequentially modifying a term, and re-verifying each time only a smaller amount of information than the whole term, still ensuring that it is well-typed.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Data types and structures; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs — Logics of programs

General Terms Theory, Languages

Keywords incrementality, type checking, logical framework, version control

1. Introduction

As programs grow and type systems become more involved, writing a correct program in one shot becomes quite difficult. On the other hand, writing a program in many correct steps is the usual practice when the time for verification is negligible; the verification tool then rechecks the entire development at each step. But this gets more tedious especially when the language in question contains proof aspects, and verification involves proof search. Some mechanisms already exist to cope with the incrementality of proofs or program development: separate compilation, interactive toplevel with undo, tactic languages; they all provide in different ways a rough approximation of the process of modifying and checking incrementally a large term.

We propose here an architecture for a generic and safe incremental type checker, a data structure for repositories of typed proofs and a language for describing proof deltas. It is based on the simple idea of sharing common subterms to avoid rechecking, and exploits encoding a derivation in a metalanguage to guarantee the well-typing of the result. This way, given a signature declaring the typing rules and an (untrusted) typing algorithm for my language of choice, I get an incremental type checker for that language. The metalanguage approach gives us the ability to encode all the aforementioned usual incrementality mechanisms in a type-safe way, and more, making our system akin to a typed version control system.

TLDI '12 January 28 2012, Philadelphia, USA.

2. Sharing-based incrementality

As a first example, let us consider a purposedly simplistic sorted language of boolean and arithmetic expressions:

$$e, e' \ ::= \ n \mid e + e' \mid e \wedge e' \mid e \leq e'$$

The algorithm to determine in a batch fashion whether the term

$$e_1 = (1 + 3 \leq 2 + 4) \wedge (8 \leq 3)$$

is well-sorted is trivial (we don't care about its evaluation here, just its well-sortedness). But what if I then change subterm $2 + 4$ in e_1 into $7 \leq 2 + 4$, to obtain e_2? Clearly, it should be verified that context $7 \leq []$ is well-sorted (it is), that $2 + 4$ "fits" into its hole (it does), that the whole expression "fits" into its new context $(1 + 3 \leq []) \wedge (8 \leq 3)$ (it does not); but the other, unchanged subterms need not be verified again. To achieve this incremental verification, the system would have to "remember" the states of the verifier in some way.

If only we had *names* (memory addresses, hashes) for enough subterms of our initial term,

$$e_1 = (\overbrace{1 + 3}^{X} \leq \overbrace{2 + 4}^{Y}) \wedge \overbrace{(8 \leq 3)}^{Z} \ ,$$

we could express concisely the change as a *delta*

$$\delta_1 = (X \leq (7 \leq Y)) \wedge Z \ ,$$

using these names to refer to unchanged subterms. If only we had *annotated* our initial term with the states of the verifier,

$$e_1 = (\overbrace{1 + 3}^{X:\mathbb{N}} \leq \overbrace{2 + 4}^{Y:\mathbb{N}}) \wedge \overbrace{(8 \leq 3)}^{Z:\mathbb{B}} \ ,$$

we would have a simple process to verify e_2 taking advantage of e_1's derivation, in $O(|\delta_2|)$: verify δ_1 as a term, retrieving the sort of names from a stored map. This suggests a data structure for a *repository* of named, annotated, verified subexpressions: a monotonously growing map, from names, or *metavariables*, to terms and types, together with a *head* metavariable identifying the top of the expression:

$$\mathcal{R} \ ::= \ X, \Delta \quad \text{where} \quad \Delta : (X \mapsto M : A) \ .$$

3. A metalanguage to encode derivations

What language are terms M and types A written into? Terms should encode our expressions with metavariables, and types A should encode the whole state of the batch verifier (here the sort). An obvious choice is to take $M \ ::= \ (e \text{ with metavariables})$ and $A \ ::= \ \mathbb{B} \mid \mathbb{N}$, but switching to another language, we'd have to redefine another repository language. Moreover, this choice is no longer expressive enough when introducing binders. Another more modular choice for this is the *metalanguage LF* [2]: it allows to specify syntax and rules of an object language as a *signature* Σ, and check terms against this signature with a generic algorithm. We'll use an increasing fragment of it. The so-called *intrinsic* style of LF signature for our expression language is:

$$\begin{aligned} \mathsf{tp} &: *, \quad \mathsf{nat} : \mathsf{tp}, \quad \mathsf{bool} : \mathsf{tp}, \quad \mathsf{exp} : \mathsf{tp} \to \mathsf{nat}, \\ \mathsf{atom} &: \mathbb{N} \to \mathsf{exp\ nat}, \\ \mathsf{plus} &: \mathsf{exp\ nat} \to \mathsf{exp\ nat} \to \mathsf{exp\ nat}, \\ \mathsf{and} &: \mathsf{exp\ bool} \to \mathsf{exp\ bool} \to \mathsf{exp\ bool}, \\ \mathsf{leq} &: \mathsf{exp\ nat} \to \mathsf{exp\ nat} \to \mathsf{exp\ bool} \end{aligned}$$

In this style, both the encoding of an expression and its sort are *terms* in the metalanguage, but the sort appears in the *type* of the encoded expression. As an example, the repository associated with expression e_1 is

$$T, \left(\begin{array}{l} X \mapsto \mathsf{plus}\ 1\ 3 : \mathsf{exp\ nat} \\ Y \mapsto \mathsf{plus}\ 2\ 4 : \mathsf{exp\ nat} \\ Z \mapsto \mathsf{leq}\ 8\ 3 : \mathsf{exp\ bool} \\ T \mapsto \mathsf{and}\ (\mathsf{leq}\ X\ Y)\ Z : \mathsf{exp\ bool} \end{array} \right)$$

The dependent nature of types in LF allows to express more complex languages. We can for example add functions, applications and variables to our expressions in a purely first-order style (using de Bruijn indices for variables) if we annotate them not only with sorts but with an environment of free variables:

$$\begin{aligned} \mathsf{exp} &: \mathsf{env} \to \mathsf{tp} \to *, \\ \mathsf{atom} &: \Pi E : \mathsf{env}.\ \mathbb{N} \to \mathsf{exp}\ E\ \mathsf{nat}, \\ \mathsf{var} &: \Pi E : \mathsf{env}.\ \Pi A : \mathsf{tp}.\ \mathsf{var}\ E\ A \to \mathsf{exp}\ E\ A, \\ \mathsf{leq} &: \Pi E : \mathsf{env}.\ \mathsf{exp}\ E\ \mathsf{nat} \to \mathsf{exp}\ E\ \mathsf{nat} \to \mathsf{exp}\ E\ \mathsf{bool}, \\ \mathsf{lam} &: \Pi E : \mathsf{env}.\ \Pi A, B : \mathsf{tp}.\ \mathsf{exp}\ (\mathsf{cons}\ A\ E)\ B \\ &\qquad \to \mathsf{exp}\ E\ (\mathsf{arr}\ A\ B) \\ &\ldots \end{aligned}$$

The encoded expressions are however very verbose: each term constructor takes as argument all these annotations. We can nonetheless make these information *implicit* in terms (but explicit in types) and let a reconstruction algorithm infer them, as in [3]. This reconstruction is language-dependent, user-provided but does not impair the safety of the system for the whole term is still checked afterwards.

LF promotes the use of lambda-tree syntax to represent binders: instead of encoding the syntax first-order, it uses the λ binder built in LF to encode binders in the object language. This style of encoding has the advantage of making the manipulation of the environment (weakening, exchange...) implicit in the deltas, but raises new challenges for the delta language and the verification process: how to share a subterm underneith a lambda? How to efficiently verify that such a delta is well-typed?

4. Expressivity

Aside from enabling to encode a large class of deductive systems safely and generically, the metalanguage approach allows to express incrementality features usually implemented in an ad-hoc manner, simply by adding new constants to the signature.

Version control Suppose we want to implement an *undo system*, storing successive versions of a closed expression of sort *bool* and able to rollback to a previous version. We add constants

$$\begin{aligned} &\mathsf{version} : *, \quad \mathsf{vnil} : \mathsf{version}, \\ &\mathsf{vcons} : \mathsf{exp\ nil\ bool} \to \mathsf{version} \to \mathsf{version} \end{aligned}$$

to the signature. The empty repository is now represented as $vnil$. Each time we have pushed a full expression M, and if S was the previous head (a version called its *ancestor*), we push vcons M S. This gives us a data structure for an undo stack, and a *commit* algorithm. But the sharing inherent to our repositories lets us actually represent *trees* of versions, by sharing common stack tails, each list head being a *branch*. Reconciling two branches' changes into a unique head is called *merging* in version control system's terminology: a merge is a version with several ancestors. We can represent merges by revising our previous addition to the signature into

$$\begin{aligned} &\mathsf{version} : *, \quad \mathsf{ancestors} : *, \quad \mathsf{anil} : \mathsf{ancestors}, \\ &\mathsf{acons} : \mathsf{version} \to \mathsf{ancestors} \to \mathsf{ancestors}, \\ &\mathsf{vcons} : \mathsf{exp\ nil\ bool} \to \mathsf{ancestors} \to \mathsf{version} \end{aligned}$$

This defines a data structure to represent (acyclic) *graphs* of versions; it is the exact data structure of repository used by version control systems Git, Monotone and Mercurial (see e.g. [1]) except that where they have directories and text files we have arbitrary typed terms.

Top-down construction While our system is based on *bottom-up* term construction, we can encode *top-down* construction common to some programming environments (e.g. Agda) and tactic-based proof assistants (e.g. Coq). The user constructs terms by successively filling *holes* with terms containing other holes. To add (linear) holes to our expressions, add constant

$$\mathsf{hole} : \Pi E : \mathsf{env}.\ \Pi A : \mathsf{tp}.\ \mathsf{exp}\ E\ A$$

to the signature. To instantiate a hole with an expression, commit the substituted term preserving sharing of subterms.

5. Architecture

We can implement this system following a layered architecture.

The *kernel* is the component in charge of verifying terms against a signature and a repository, and updating this repository. It supports two basic operations:

- $\mathsf{push}_\Sigma(\mathcal{R}, M)$ checks a small part M of a larger term against Σ in $\mathcal{R}$, synthetizes its type A, chooses a fresh metavariable X for M and returns $\mathcal{R}[X \mapsto M : A]$ and X.
- $\mathsf{pull}_\Sigma(\mathcal{R}, X)$ returns the term M associated with X in $\mathcal{R}$ recursively: all metavariables are unfolded to their definitions.

The *slicer* is the component in charge of slicing a term M into many terms, pushing them to the repository to enable future sharing, and adding version markers. It supports operations:

- $\mathsf{commit}_\Sigma(\mathcal{R}, M)$ pushes vcons M (acons X anil) to $\mathcal{R}$ in several push() operations, where X is the current head.
- $\mathsf{merge}_\Sigma(\mathcal{R}, M, Y)$ pushes vcons M (acons X (acons Y anil)) to $\mathcal{R}$. Note that it doesn't actually perform the merge, it simply commits a previously computed merge node with value M.

The *reconstructor* performs the reconstruction of the derivation (an M) from the initial expression (an e), given the derivations for its metavariables (an $\mathcal{R}$), and the expected type (exp nil bool), and commits M.

Finally, the *compressor* computes a delta e' from a metavariable-free expression e by recognizing equal subterms in $\mathcal{R}$. This can be achieved by *hash-consing*.

References

[1] S. Chacon. Git community book. *The Git Community*, 2009. URL `http://book.git-scm.com/`.

[2] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.

[3] G. Necula and P. Lee. Efficient representation and validation of proofs. In *Logic in Computer Science, 1998. Proceedings. Thirteenth Annual IEEE Symposium on*, pages 93–104. IEEE, 1997.