Certifying, incremental type checking

Matthias Puech[1,2] and Yann Régis-Gianas[1]

[1] Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126 CNRS, πr^2, INRIA Paris-Rocquencourt, F-75205 Paris, France
[2] Department of Computer Science, University of Bologna, 40126 Bologna, Italy

Abstract. A lightweight way to design a trusted type-checker is to let it return a *certificate* of well-typing, and check it a posteriori (for instance Agda and GHC adopt this architecture internally). Most of the time, these type-checkers are confronted sequentially with very similar pieces of input: the same program, each time slightly modified. Yet they rebuild the entire certificates, even for unchanged parts of the code. Instead of discarding them between runs, we can take advantage of these certificates to improve the reactivity of type-checking, by feeding parts of the certificates for unchanged parts of the original program back into the new input, thus building an incremental type-checking.
We present a language-independent framework making possible to turn a type system and its certifying checking algorithm into a safe, incremental one. It consists in a model of incrementality by sharing, a generic data structure to represent reusable derivations based on contextual LF, and a general-purpose language to write certifying programs.

Introduction

Verifying the well-typing of real-world programs is a critical matter: on it relies their safety, and it encourages good programming practices: modularity, early bug catching.... But it is a complex operation, for the compiler designer and for the human operator:

For the compiler designer Type systems are usually presented in a declarative manner as a set of inference rules, together with proofs that all programs accepting a derivation in this system respect a certain dependability claim, the famous "progress and preservation" pair. Yet, actual algorithms of verification often differ very much from their declarative description. This is of course true when programs are provided with little type annotations ("type inference"), in which case the missing information has to be algorithmically synthesized, but is also true in many cases for mere "type checking", or explicit "church-style" calculi. Take for example dependent type systems, relying on definitional equality on types, or type systems with subtype polymorphism: both feature a rule of the form

$$\frac{\vdash M : A \qquad A \leq B}{\vdash M : B}$$

(for a given notion of $\leq$) which is not syntax-directed. The implementer has to turn the declarative system into an algorithm, and convince himself that both are equivalent. This task is highly non-trivial and non-modular: for instance, how to infer the type of a System T recursor $\mathsf{rec}(M, N, x\ y.\ P)$, in presence of subtyping? The problem is that we have to guess a type A more general than that of N and P, while giving that type to variable y. A well-known algorithm if the subtyping relation forms a complete lattice is to iterate the typing of P with the join of the previously computed and the new type until a fixpoint is reached.

The most common option is to formally prove the equivalence, i.e. that the (typing) relation is actually a function. Although these proofs can be difficult to obtain, it is the most satisfactory because it allows to take only the result of the algorithm into consideration. But let us choose another path: derivations in the declarative systems, if they cannot always be easily inferred, can be *verified* if provided by an oracle. We can tell cheaply if it is wrong by verifying the witness typing derivation it provided. Let us then imagine a type-checker to be the pair of *(i)* an untrusted procedure $\uparrow M$ to infer a typing derivation out of program M; it does not have to be complete, or even sound, and *(ii)* a trusted and fast kernel $\mathsf{check}(\mathcal{D})$ to verify these derivations w.r.t. a given set of inference rules Σ. This is a *certifying* scheme (see [1]). If $\mathsf{check}(\uparrow M) = \mathsf{true}$ then there really *is* a derivation $\mathcal{D}$ for M, but the converse is not necessarily true: function $\uparrow$could fail, or return an incorrect derivation. Explicitly generating the derivation in the declarative system has the following advantages over the first approach:

- It is lightweight: the untrusted algorithm is usually shorter than the corresponding equivalence proof.
- Compilers often need to propagate type information further in the chain of transformation; the typing derivation produced by $\uparrow$ being explicit, it could constitute the data structure being exploited.
- It inherits benefits from certifying approaches like proof-carrying code [2]: certificates can be transmitted with the program, saving regeneration time and avoiding to disclose the generation program.

... but there is more:

For the programmer As type systems become richer, it becomes increasingly difficult for the programmer to write a well-typed program in one try, compile it and run it: writing a program becomes a tight and non-linear interaction between the human and the type-checker involving constant experiments, fixes, etc. If this interaction is possible for short programs and fast checkers, it becomes increasingly hindered by the latency of complete rechecks: even if the modifications made to the program between two interactions are small, the whole program is usually rechecked. This is especially true for languages embedding formal verification aspects, like proof assistants: there, tight interaction with a type or proof checker is unavoidable due to the difficulty of writing correct proofs without guidance from the system, and the time taken by many tools to infer easy parts of the proofs (proof search). The constant modification of the source makes it necessary to observe *incrementally* the effect of a small change on the whole edifice: we

need to take advantage of the results of previous checks in order to build the result for a modified version of the program.

Several generic methods have been devised for turning a batch program into an incremental one [3–5]. Unfortunately, they suffer from their genericity on the particular problem of type checking:

- First, they ignore the higher-order nature of program syntax, and the kind of structural quotients we make on syntax trees with binders and environments (α-equivalence, weakening...): a hypothetical derivation of $\vdash x + 2 : \mathsf{nat}$ is still valid when put under a binder $\lambda y.\ (x + 2) - y$ and should not be regenerated.
- Since it is easy to track edits on a very large programs (trace of the editor), we may want to feed the type checker only with a small *delta* on the previous version and not the whole program, thus saving the time to recognize already checked parts;
- Finally, there is no way to a posteriori validate the result of the incremental process: the user has to trust not only the type-checking algorithm but also its transformation into an incremental one, which increases the size of the trusted base.

Now, if we consider as above a type-checker as a derivation-generating procedure, and storing these derivations between calls to the procedure, we can reuse already-built subderivations for subterms common to an old, checked version and a new, to-be-checked version, and graft together these pieces of derivation to form a new, large one for the new version of the program, saving the regeneration of all equal subterms' derivations in compatible contexts. This way we constantly maintain the verifiable witness of well-typing for the program, and take advantage of the quotients made on syntax with binders.

We can thus add to the list of advantages begun earlier that explicitly generating typing derivations makes possible a form of safe incrementality by sharing common subderivations.

In this paper, we present a framework to incrementally generate typing certificates represented in the LF logical framework, and implemented as an ML library. We specifically propose:

- a simple language for representing programs and proof deltas, derived from contextual LF [6];
- a computational language to write untrusted derivation-generating type checkers, based on a notion of function inverse to refer to already-computed values;
- a generic algorithm to evaluate these type-checkers on program deltas and verify on-the-fly the generated incremental certificates against a *repository* of already constructed and checked derivations.

1 Using the framework

1.1 As a programmer

A student is learning about System $\mathsf{T}_{<:}$ — a simply typed λ-calculus with natural numbers nat coming in two sets even and odd, introduction o and $\mathsf{s}(M)$, elimination ($\mathsf{rec}(M, N, x\ y.\ P)$), and subtyping on these types. He (or she) wants to write a program computing $(2+2)\times 3$ in that system. He knows that sometimes the arguments of rec are quite hard to get correct, but $\mathsf{T}_{<:}$ is a typed language; he wants to take advantage of this to build his program incrementally using the teacher's provided incremental verifier. As a first attempt, he wants to verify $P_1 = \mathsf{rec}(\mathsf{s}(\mathsf{o}), \mathsf{s}(\mathsf{o}), x\ y.\ \mathsf{s}(x))$: he sends the command $\uparrow P_1$ to the system. The $\uparrow$ operator takes a well-formed program and tries to construct a valid typing derivation for its argument in the current, empty context. The system succeeds and returns a typing derivation $\mathcal{D}_1$ of $\vdash P_1 : \mathsf{nat}$, valid in an empty context.

The student then decides to factorize a definition for *add* out of his code:

$$\textbf{let}\ add = \lambda x : \mathsf{nat}.\ \lambda y : \mathsf{nat}.\ \mathsf{rec}(x, y, u\ v.\ \mathsf{s}(u))\ \ \textbf{in}\ \ add\ (\mathsf{s}(\mathsf{o}))\ (\mathsf{s}(\mathsf{o}))\ .$$

But he (or his text editor) realizes that some parts of this program have not changed: both $\mathsf{s}(\mathsf{o})$, and the function body $\mathsf{s}(u)$. He could reuse the pieces of derivation just built. Let us call these respectively $\mathcal{D}_2$, $\mathcal{D}_3$ and $\mathcal{D}_4$. $\mathcal{D}_4$ is hypothetical: it depends on the hypothesis H that $\vdash u : \mathsf{nat}$. Provided there is an operator $\downarrow\mathcal{D}$ from the derivation to the program it types, what really needs to be sent to the system is:

$$\uparrow(\textbf{let}\ add = \lambda x : \mathsf{nat}.\ \lambda y : \mathsf{nat}.\ \mathsf{rec}(x, y, u\ v.\ \downarrow\mathcal{D}_4[H/\uparrow u])\ \ \textbf{in}\ \ add\ \downarrow\mathcal{D}_2\ \downarrow\mathcal{D}_2)\ ,$$

that is, only the changed subterms of the program: this is a *delta*. Derivation $\mathcal{D}_2$ used to live in the empty context, we are then free to use it in the new context where $\vdash add : \mathsf{nat} \to \mathsf{nat}$, because System T enjoys the *weakening* property; $\mathcal{D}_4$ depending on a derivation $\vdash x : \mathsf{nat}$, we instantiate this hole with the derivation $\uparrow u$ inferred recursively for the new recursive argument u. $\mathcal{D}_2$ derives $\vdash \mathsf{s}(\mathsf{o}) : \mathsf{odd}$, but *add* awaits two nat arguments. When the system generates the derivation for this application, it inserts a coercion $\mathsf{odd} \leq \mathsf{nat}$ at these two points. Taking advantage of $\mathcal{D}_2$ and $\mathcal{D}_4$ the system succeeds with a new derivation made out of these, without having to regenerate them.

Let us call $\mathcal{D}_5$ the subderivation generated for the definition of *add*, $\mathcal{D}_6$ for the arguments $\downarrow\mathcal{D}_2\ \downarrow\mathcal{D}_2$ and $\mathcal{D}_7$ for the application $add\ \downarrow\mathcal{D}_6$. Even if it occurs originally in the scope of *add*, $\mathcal{D}_6$ does not depend on the hypothesis H that $\vdash add : \mathsf{nat} \to \mathsf{nat}$ because of the *strengthening* property. Contrarily, $\mathcal{D}_7$ does: each time we want to reuse it, we will have to provide a proof of this fact.

The student then provides a definition for multiplication, and writes the operation $(2 + 2) \times 3$:

$$\begin{aligned}\uparrow(\ &\textbf{let}\ add = \downarrow\mathcal{D}_5\ \ \textbf{in}\ \ \textbf{let}\ mul = \lambda x : \mathsf{nat}.\ \lambda y : \mathsf{nat}.\ \mathsf{rec}(\mathsf{o}, y, z\ t.\ add\ x\ z)\ \ \textbf{in}\\ &\quad mul\ (\downarrow\mathcal{D}_7[H/\uparrow add])\ (\mathsf{s}(\downarrow\mathcal{D}_2)))\ .\end{aligned}$$

Here again, he took care of reusing all derivations for equal subterms. In particular, $\mathcal{D}_7$ expects a proof that the variable *add* is of type $\mathsf{nat} \to \mathsf{nat}$; he provides $\uparrow add$ meaning: at this point we will have generated (or reused in this case) the derivation of *add*, so we just have to use it here. $\mathcal{D}_2$ was deriving $\vdash \mathsf{s}(\mathsf{o}) : \mathsf{nat}$ (with a coercion since it was used as an argument to *add*); the system thus infers the more general type nat for $\mathsf{s}(\downarrow \mathcal{D}_2)$.

Discussion We choose here to represent deltas as usual terms with variables x, y and special, *meta*-variables referring to already-computed results $\mathcal{D}_i$. Commands sent to the system consists of terms, possibly containing special typed operators having a computational content: $\uparrow$ takes a term M and produces the (dependent) pair of a type A and a derivation of the judgment $\vdash M : A$ in the current context; $\downarrow$ does the exact converse: it takes a derivation $\vdash M : A$ and projects it back as the term M. The successful result of a command execution is a derivation where each subderivation has been given a unique metavariable name for later reuse (a *repository*). As subderivations can have free variables, we close them by a local environment that must be instantiated by a *substitution*.

Although this small example may seem anecdotal, the interest of such a mechanism becomes clear on very large programs: then the cost of rechecking well-typing of a small modification amounts to recheck the path in the term from the root to the changed part, and the changed part itself.

Seeing type-checking as issuing a completely explicit typing derivation enables thus to think of the relationship between the input and the output. Feeding subresults of the output $\mathcal{D}_i$ back into the input P_i is a well-known technique for handling changes in the input: combined with a way to automatically recognize already-computed input, it is *memoization*, a general-purpose technique for incremental computation that has been extensively studied and combined with other techniques [4, 3]. However, it is best understood in the case of purely first-order data, and to our best knowledge remains mainly unexplored for higher-order structures containing binders, like programs and proofs.

A memoized type-checker stores in a table bindings from its input (the program) to its output (the derivation), and returns automatically the stored output if the current input matches a binding in the table. It thus performs two operations at once: output reuse and recognition of already-seen input. Recognition — that is from a modified program, generate a delta from previous versions — is in the higher-order case a complex task that we hope to address in the future; we tackle here the verification of output reuse: given a delta, construct a new derivation and verify it.

1.2 As a type system designer

Earlier, the teacher is preparing the tool for the students. He (or she) provides the abstract syntax, the typing rules and the untrusted typing algorithm in order to build the incremental type checker. How should he write the algorithm? We want it to return a derivation, and take advantage of a higher-order representation

of the syntax, so it should not be necessary to thread any explicit environment. The definition should then start with:

$$\uparrow \ : \ \Pi M : \mathsf{tm}.\ \Sigma A : \mathsf{tp}.\ (\vdash M : A) =$$

Function $\uparrow$ takes a term M and produces the pair of an inferred type A and a derivation that M has type A. Its code will be untyped, but the teacher provides this type so that the checker can verify that each call is fed and returns values of the right type: it is its *contract* so to say. He first decomposes term M and treats the easy cases of o and s. The code continues with:

$$\begin{array}{l}
\lambda M.\ \mathbf{case}\ M\ \mathbf{of} \\
\quad \mid \mathsf{o} \to \langle \mathsf{even}, \dfrac{}{\vdash \mathsf{o} : \mathsf{even}} \rangle \\
\quad \mid \mathsf{s}(M) \to \mathbf{case}\ \uparrow M\ \mathbf{of} \\
\qquad \mid \langle \mathsf{even}, \mathcal{D} \rangle \to \langle \mathsf{odd}, \dfrac{\mathcal{D}}{\vdash \mathsf{s}\ M : \mathsf{odd}} \rangle \\
\qquad \mid \langle \mathsf{odd}, \mathcal{D} \rangle \to \langle \mathsf{even}, \dfrac{\mathcal{D}}{\vdash \mathsf{s}\ M : \mathsf{even}} \rangle \\
\qquad \mid \langle \mathsf{nat}, \mathcal{D} \rangle \to \langle \mathsf{nat}, \dfrac{\mathcal{D}}{\vdash \mathsf{s}\ M : \mathsf{nat}} \rangle
\end{array}$$

We write $\langle \cdot, \cdot \rangle$ for the constructor of Σ-types. Patterns do not have to be exhaustive since we are in an untrusted setting. For conciseness, we use case failure for signaling typing error. There are three cases for $\mathsf{s}(M)$ in the system, corresponding to the three rules in the declarative system. For the application case, we rely on the well-known property that we may need to insert a coercion from the actual type of the argument to its expected type:

$$\begin{array}{l}
\mid M\ N \to \\
\quad \mathbf{let}\ \langle A_1 \to B, \mathcal{D}_1 \rangle = \uparrow M\ \mathbf{in}\ \mathbf{let}\ \langle A_2, \mathcal{D}_2 \rangle = \uparrow N\ \mathbf{in}\ \mathbf{let}\ \mathcal{D}_{\leq} = A_1 \leq A_2\ \mathbf{in} \\
\quad \mathbf{case}\ \mathcal{D}_{\leq}\ \mathbf{of} \\
\qquad \mid \dfrac{}{\vdash A \leq A} \to \langle B, \dfrac{\mathcal{D}_1 \qquad \mathcal{D}_2}{\vdash M\ N : B} \rangle \\
\qquad \mid _ \to \langle B, \dfrac{\mathcal{D}_1 \qquad \dfrac{\mathcal{D}_2 \qquad \mathcal{D}_{\leq}}{\vdash N : A_1}}{\vdash M\ N : B} \rangle
\end{array}$$

The **let** construct is just syntactic sugar for a one-branch **case**; we use an auxiliary function

$$\leq \ : \ \Pi A : \mathsf{tp}.\ \Pi B : \mathsf{tp}.\ \vdash A \leq B = \ldots$$

trying to construct a subtyping derivation for its argument, or failing with an error. Notice the slight optimization to the size of the derivation: if the types A_1 and A_2 are the same syntactically, we do not generate a subtyping rule. This is done by matching on the returned derivation $\mathcal{D}_{\leq}$, in the case of the Refl rule.

Then comes the abstraction case $\lambda x : A.\ M$. The difficulty is that the recursive call $\uparrow M$ must be done in an enlarged environment where $\vdash x : A$. Otherwise the

check that the open derivation $\mathcal{D}$ returned by the recursive call is correct would fail (x is unknown). We thus introduce a new construct **case** M **in** Γ **of** ... (**case** M **of** ... being syntactic sugar for **case** M **in** $\cdot$ **of** ...). Even then, we would have no way of synthesizing a variable x's type anymore ($\mid x \to ???$). We therefore substitute all occurrences of x with the *output* of function $\uparrow$ on it (since it is known at this point), i.e. a pair of A and $\mathcal{D}_1$. For this substitution to be well-typed, we now need to coerce this output to a valid input (a term). We introduce an operator which is in some sense the *inverse* of $\uparrow$: operator $\downarrow$ maps a pair of a type and a derivation to the term it types.

$$\mid \lambda x : A.\ M \to \mathbf{let}\,\langle B, \mathcal{D}\rangle\,\mathbf{in}\,\mathcal{D}_x : (\vdash x : A) = \uparrow M[x/\downarrow\langle A, \mathcal{D}_x\rangle]\ \mathbf{in}$$

At this point, we get back B and $\mathcal{D}$ from the recursive call. We know that B cannot depend on $\mathcal{D}_x$ (because a System $\mathsf{T}_{<:}$ type is always closed), but $\mathcal{D}$ can: it is not *per se* a hypothetical derivation but an *open* derivation where a special judgement named $\mathcal{D}_x$ can occur. To return the complete derivation of $\vdash \lambda x : A.\ M : A \to B$, we must then close, or "hypothesize" $\mathcal{D}$ by $\mathcal{D}_x$, and then apply rule Lam:

$$\langle A \to B, \frac{\begin{array}{c}[\mathcal{D}_x]\\ \mathcal{D}\end{array}}{\vdash \lambda x.\ M : A \to B}\rangle$$

Concerning the treatment of environments, the case of $\mathsf{rec}(M, N, x\ y.\ P)$ is similar: we first compute recursively the derivations $\mathcal{D}_M$ and $\mathcal{D}_N$ which should be well-typed in the current environment, and then $\mathcal{D}_P$ in the enlarged environment, with variables x and y in P substituted with the (future) result of $\uparrow$.

One problem remains with subtyping: terms $N : A_N$ and $P : A_P$ should be coercible to the same type A, but P should be typed under $\vdash y : A$, which is not known before typing P. A trick to break the loop is to iterate the typing of P with a more and more general type, until we reach a fixpoint. We reach a fixpoint if there is a unique more general type, in our particular case of subtyping relation, this means iterating only twice: we will then reach nat.

$$\begin{array}{l}
\mid \mathsf{rec}(M, N, x\ y.\ P) \to \\
\quad \mathbf{let}\,\langle A_M, \mathcal{D}_M\rangle = \uparrow M\ \ \mathbf{in}\ \ \mathbf{let}\,\mathcal{D}_{A_M} = A_M \le \mathsf{nat}\ \ \mathbf{in}\ \ \mathbf{let}\,\langle A_N, \mathcal{D}_N\rangle = \uparrow N\ \ \mathbf{in} \\
\quad \mathbf{let}\,\langle A_P, \mathcal{D}_P\rangle\,\mathbf{in}\,(\mathcal{D}_x : (\vdash x : \mathsf{nat}), \mathcal{D}_y : (\vdash y : A_N)) = \\
\qquad \uparrow P[x/\downarrow\langle \mathsf{nat}, \mathcal{D}_x\rangle, y/\downarrow\langle A_N, \mathcal{D}_y\rangle]\,\mathbf{in} \\
\quad \mathbf{let}\,\langle A, \langle \mathcal{D}_{A_N}, \mathcal{D}_{A_P}\rangle\rangle = A_N \sqcap A_P\ \ \mathbf{in} \\
\quad \mathbf{let}\,\langle _, \mathcal{D}_P\rangle\,\mathbf{in}\,(\mathcal{D}_x : (\vdash x : \mathsf{nat}), \mathcal{D}_y : (\vdash y : A)) = \\
\qquad \uparrow P[x/\downarrow\langle \mathsf{nat}, \mathcal{D}_x\rangle, y/\downarrow\langle A, \mathcal{D}_y\rangle]\,\mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad [\mathcal{D}_x][\mathcal{D}_y] \\
\qquad \langle A, \dfrac{\dfrac{\mathcal{D}_M \quad \mathcal{D}_{A_M}}{\vdash M : A} \qquad \dfrac{\mathcal{D}_N \quad \mathcal{D}_{A_N}}{\vdash N : A} \qquad \dfrac{\mathcal{D}_P \quad \mathcal{D}_{A_P}}{\vdash P : A}}{\vdash \mathsf{rec}(M, N, x\ y.\ P) : A}\rangle
\end{array}$$

Again, we use a new auxiliary function

$$\sqcap\ :\ \Pi A : \mathsf{tp}.\ \Pi B : \mathsf{tp}.\ \Sigma C : \mathsf{tp}.\ (\vdash A \le C) \times (\vdash B \le C) = \ldots$$

computing the sup of two types A and B, and returning proofs of subtyping. Note that we could have performed the optimization seen in the application case as well.

The type-checker is now done. There is no case for variables, as we will not ever meet this case: we already substituted all variables with their result derivation. Evaluation only needs to make sure that in this case, the derivation will be returned, i.e. that equation $\uparrow\downarrow\langle A, \mathcal{D}\rangle = \langle A, \mathcal{D}\rangle$ holds.

Note that $\downarrow$ can be derived automatically from the type of $\uparrow$: we will call it its inverse, since it maps the return type to the argument. It has definition:

$$\downarrow \ : \ \Pi M : \mathsf{tm}.\ \Pi A : \mathsf{tp}.\ \Pi \mathcal{D} : (\vdash M : A).\ \mathsf{tm} = \lambda M.\ \lambda A.\ \lambda \mathcal{D}.\ M$$

where argument M is left implicit (it can be inferred).

Discussion We exploit an idea similar to run-time *contracts* [7]: there is no static guarantee that computations will not return ill-typed values, but arguments and results are checked at run-time at the boundaries of defined functions. This allows to omit formal justifications, like the fact that we need to iterate twice the typing of P in $\mathsf{rec}(M, N, x\ y.\ P)$, or that B cannot depend on $\mathcal{D}_x$ in the λ case, while detecting errors early.

The idea of untrustedly generating certificates verified *a posteriori* is well-known to LCF-style proof assistants: during manual proof search, the interaction is driven by *tactics*: they are untrusted programs generating pieces of proofs, assembled by other, higher-order tactics; verification of the generated proofs is done when the evaluation of all tactics is complete. The differences with our mechanism are two-fold: first, tactics usually take one (or several) *goal*(s) as input, i.e. holes in a proof, whereas our functions can take any term; secondly, proof-check is usually done at the end of the evaluation, which can lead to less tractable bugs; by making evaluation and checking mutually recursive, we choose to detect errors earlier in these witnesses: exactly at the point where they are generated.

Introducing the "inverse" of the derivation inference function has two purposes: as we saw in 1.1, it allows to refer to already-constructed derivations when writing deltas. It also permits to abstract from the concrete environment when writing the type-checker itself, that one must thread throughout the process in a traditional, first-order type-checker. It is a generalization of the so-called *context-free* typing presented in [8, chap. 4] where a special term construct $\{x : A\}$ denoting free variables is added to the syntax of terms, and rules

$$\frac{\vdash M[x/\{x : A\}] : B}{\vdash \lambda x : A.\ M : A \to B} \qquad \text{and} \qquad \frac{}{\vdash \{x : A\} : A}$$

replace the usual ones. The difficulty with this technique, particularly in a dependent setting, is to ensure that term $\{x : A\}$ reduces to x so that they are considered equal, i.e. we can strip off the annotations, while not forgetting them too soon: otherwise we would hit the unhandled case $\uparrow x$. This requires evaluating the program deltas with a carefully designed strategy (see 2.3).

$$
\begin{array}{rcll}
K & ::= & \Pi x : A.\ K \mid * & \text{Kind} \\
A & ::= & \Pi x : A.\ A \mid P & \text{Type family} \\
P & ::= & \mathsf{a}\ S & \text{Atomic type} \\
M & ::= & \lambda x.\ M \mid F & \text{Canonical object} \\
F & ::= & H\ S \mid X[\sigma] & \text{Atomic object} \\
H & ::= & x \mid \mathsf{c} \mid \boldsymbol{f} & \text{Head} \\
S & ::= & \cdot \mid M\ S & \text{Spine} \\
\sigma & ::= & \cdot \mid \sigma, x/M & \text{Parallel substitution} \\
\Gamma & ::= & \cdot \mid \Gamma, x : A & \text{Environment}
\end{array}
$$

Fig. 1. Syntax of SLF

2 Design of the framework

2.1 The representation language

The emission of certificates by untrusted applications first raises the question of the *data structure* to use for these certificates. The Edinburgh Logical Framework [9], or LF for short, is one of the well-accepted solutions to this question. It is a dependently typed lambda-calculus for *representing* higher-order terms and derivations [10]. The user gives a *signature* defining abstract syntax and inference rules of his object language, and gets an algorithm to verify terms and derivations expressed in the LF syntax.

To write deltas by sharing subterms, we need to be able to refer to any subterm. For this, we first extend LF with contextual metavariables (written X, Y) standing for open terms, following [6]. Secondly, we make sure that every subterm is named by a metavariable (an operation called *slicing*). We thus introduce Sliced LF (SLF, see Fig. 1): it is an extension of *spine canonical* LF [11] enriched with metavariables which are references to open terms as in [6].

In this variant, only η-long and β-normal objects have an existence. Canonicity for β-reduction is enforced in the syntax and canonicity for η-expansion will be enforced by typing, following [12]. Besides, application of a *head* (i.e. a variable, constant, or function symbol) is n-ary (i.e. to a *spine* of objects) as opposed to the more standard binary application: this eases the definition of substitution and η-long forms, and corresponds to a focused sequent calculus known as LJT [13]. We differ the discussion of function symbols $\boldsymbol{f}$ to section 2.2. A *value* is an object containing no function symbols. The definition of hereditary substitution $M[\sigma]$ is standard, and it is eluded here.

We distinguish syntactically between *checkable* and *inferrable* objects (resp. *canonical* M and *atomic* F), as in e.g. [14]. Canonical objects are checked against a given type, whereas we can synthesize the type of atomic objects; the coercion from canonical to atomic objects triggers the equality check between synthesized and checked types.

As is customary, we adopt the Barendregt convention for naming bound variables, write $A_1 \to A_2$ for $\Pi x : A_1.\ A_2$ when $x \notin \mathsf{FV}(A_2)$, X for $X[\cdot]$, and adopt indifferently list, map or set notations for σ and Γ. The identity substitution $\mathsf{id}(\Gamma)$ is a notation for $\{x/x \mid x \in \mathrm{dom}(\Gamma)\}$. Metavariables $X[\sigma]$ stand for open objects with free variables in $\mathrm{dom}(\sigma)$. To each use of them is attached a substitution σ defining these free variables. We write $\mathsf{FMV}(M)$ for the set of metavariables in M and call an object M *metaclosed* if $\mathsf{FMV}(M) = \emptyset$.

Definition 1. *A* repository $\mathcal{R}$ *is a finite map from metavariable to triplets of environments, atomic objects and atomic types (i.e. judgments of well-typing), along with a distinguished metavariable, corresponding to the* tip *of the term it contains:*

$$\mathcal{R} \ : \ (X \mapsto (\Gamma \vdash F : P)) \times X[\sigma]$$

If $\mathcal{R} = (\Delta, X[\sigma])$, we write $\hat{\mathcal{R}}$ for the tip $X[\sigma]$, and $\mathcal{R}(X[\sigma])$ for $M[\sigma]$ if $\Delta(X) = (\Gamma \vdash M : A)$. The algorithm in 2.3 will maintain two invariants: a repository $\mathcal{R}$ is *acyclic*, (for all X in $\mathcal{R}$, the fixpoint of $\mathsf{FMV}(\mathcal{R}(X))$ does not contain X) and all F and P in it are values.

Definition 2. *We define the partial* checkout *operation* $\mathsf{co}(\mathcal{R})$ *(resp.* $\mathsf{co}_{\mathcal{R}}(M)$*,* $\mathsf{co}_{\mathcal{R}}(S)$*) from a repository (resp. object, spine) to a metaclosed object as unfolding the definitions of all metavariables recursively:*

$$\begin{aligned}
\mathsf{co}(\mathcal{R}) &= \mathsf{co}_{\mathcal{R}}(\hat{\mathcal{R}}) & \mathsf{co}_{\mathcal{R}}(X[\sigma]) &= \mathsf{co}_{\mathcal{R}}(\mathcal{R}(X[\sigma])) \\
\mathsf{co}_{\mathcal{R}}(\lambda x.\ M) &= \lambda x.\ \mathsf{co}_{\mathcal{R}}(M) & \mathsf{co}_{\mathcal{R}}(H\ S) &= H\ (\mathsf{co}_{\mathcal{R}}(S)) \\
\mathsf{co}_{\mathcal{R}}(M\ S) &= \mathsf{co}_{\mathcal{R}}(M)\ \mathsf{co}_{\mathcal{R}}(S) & \mathsf{co}_{\mathcal{R}}(\cdot) &= \cdot
\end{aligned}$$

Example 1. The repository $((X \mapsto \cdot \vdash \mathsf{lam}\ \lambda x.\ Y[u/T[w/x]] : \mathsf{tm}, Y \mapsto u : \mathsf{tm} \vdash \mathsf{lam}\ \lambda y.\ Z[u/u, v/y] : \mathsf{tm}, Z \mapsto u : \mathsf{tm}, v : \mathsf{tm} \vdash \mathsf{app}\ v\ u : \mathsf{tm}, T \mapsto x : \mathsf{tm} \vdash \mathsf{app}\ x\ \mathsf{o} : \mathsf{tm}), X[])$ checks out as the metaclosed atomic object $\mathsf{lam}\ \lambda x.\ \mathsf{lam}\ \lambda y.\ \mathsf{app}\ y\ (\mathsf{app}\ x\ \mathsf{o})$.

2.2 The computation language

Secondly, we introduce the CL language to compute over our SLF objects (Fig. 2). We can construct atomic objects out of others, and deconstruct them by case analysis. The definition T of a function always begins with a series of λ (as many as there are arguments in its type).

This language is untyped, but types are dynamically checked: there is no static guarantee that object $\boldsymbol{f}\ x$ produced by a function $\boldsymbol{f} : A_1 \to A_2 = T$ will have type A_2, nor that pattern matching is exhaustive. It is also non-terminating: we can define a recursive function $\boldsymbol{f} : \mathsf{a} \to \mathsf{a} = \lambda x.\ \boldsymbol{f}\ x$. Nevertheless, we want to maintain the invariant that each time an object goes in (resp. out) of a function $\boldsymbol{f} : A_1 \to A_2$, it is checked against the type of this function (A_1, resp. A_2). You can think of interpreted functions as mere black boxes taking and producing objects, and being able to feed objects to other black boxes; yet the wires linking

$$
\begin{array}{lll}
T & ::= & \lambda x.\ T \mid U & \text{Term} \\
U & ::= & F \mid \mathbf{case}\ U\ \mathbf{in}\ \Gamma\ \mathbf{of}\ C & \text{Atomic term} \\
C & ::= & P \to U \mid C \mid P \to U & \text{Branches} \\
P & ::= & H\ x\ \ldots\ x & \text{Pattern}
\end{array}
$$

Fig. 2. Syntax of CL

them are labeled with types, and objects circulating in them are checked against these types.

Besides constants, the signature defines *interpreted functions* $\boldsymbol{f}$, $\boldsymbol{g}$. Each function comes with its type and its code.

Definition 3. *A* signature *declares object and type constants and functions to be used in objects.*

$$\Sigma\ ::=\ \cdot \mid \Sigma, \mathsf{a} : K \mid \Sigma, \mathsf{c} : A \mid \Sigma, \boldsymbol{f} : A = T \qquad \textit{Signature}$$

Definition 4. *To each function* $\boldsymbol{f} : A = T$ *in* Σ *we associate a family of* inverses $\boldsymbol{f}^n : (A)^n = \pi_n(A)$ *for* $n \in \mathbb{N}$*, where type* $(A)^n$ *and term* $\pi_n(A)$ *are defined recursively:*

$$
\begin{array}{rlrl}
(\Pi x : A_1.\ A_2)^{n+1} &= \Pi x : A_1.\ (A_2)^n & (\Pi x : A_1.\ A_2)^0 &= \Pi x : A_1.\ (A_2)_{A_1} \\
(\Pi x : A_1.\ A_2)_A &= \Pi x : A_1.\ (A_2)_A & (P)_A &= \Pi x : P.\ A \\
\pi_{n+1}(\Pi x : A_1.\ A_2) &= \lambda x.\ \pi_n(A_2) & \pi_0(\Pi x : A_1.\ A_2) &= \lambda y.\ \pi^y(A_2) \\
\pi^y(\Pi x : A_1.\ A_2) &= \lambda x.\ \pi^y(A_2) & \pi^y(P) &= y
\end{array}
$$

Functions $(A)^n$ and $\pi_n(A)$ use auxiliary functions $(A)_B$ and $\pi^y(A)$ to put the n-th argument at the end of the type and term.

Example 2.
- Function $\boldsymbol{infer} : (\Pi M : \mathsf{tm}.\ \mathsf{sig}\ M) = T$ (called $\uparrow$ above, with sig the LF encoding of the previous Σ-type) has one inverse $\boldsymbol{infer}^0 : (\Pi M : \mathsf{tm}.\ \mathsf{sig}\ M \to \mathsf{tm}) = \lambda x.\ \lambda y.\ x$ (called $\downarrow$ above).
- Function $\boldsymbol{equal} : \Pi M : \mathsf{tm}.\ \Pi N : \mathsf{tm}.\ \mathsf{eq}\ M\ N = T'$ has two inverses: $\boldsymbol{equal}^0$ and $\boldsymbol{equal}^1$ of the same type $\Pi M : \mathsf{tm}.\ \Pi N : \mathsf{tm}.\ \mathsf{eq}\ M\ N \to \mathsf{tm}$ with code respectively $\lambda m.\ \lambda n.\ \lambda h.\ m$ and $\lambda m.\ \lambda n.\ \lambda h.\ n$.

2.3 The evaluation algorithm

The evaluation algorithm, called *commit* $\mathsf{commit}_{\mathcal{R}}(F)$ takes an atomic object F (possibly containing function symbols to evaluate) and a repository $\mathcal{R}$, and produces a new repository $\mathcal{R}'$ which is a strict extension of $\mathcal{R}$ with a different tip, and which checks out to the value of F. It has three roles, which are necessarily interleaved:

– it evaluates M by reducing function symbols,
– it type-checks all values fed to and produced by functions w.r.t. $\mathcal{R}$,
– it slices those values. This requires threading the map of $\mathcal{R}$ throughout evaluation and typing.

We now turn to the bidirectional typed reduction and slicing algorithm (Fig. 3, 4, 5), presented as an inference system. All judgments are of the form $\mathcal{U} \vdash_L \mathcal{V} \Rightarrow \mathcal{W}$ where $\mathcal{U}$, $\mathcal{V}$ and L are inputs, are directed by the syntax of $\mathcal{V}$ and L, and $\mathcal{W}$ is the output. They are parameterized by a constant and implicit signature Σ.

Label L is a mode influencing the strategy of evaluation of defined function applications. They range over $\top$, $\bot$ and $\boldsymbol{f}$, meaning respectively strong reduction, pure checking with no reduction and call-by-name, head reduction. The purpose of the $\boldsymbol{f}$ annotation is shown in rule FAppEvalInv: it detects the special redex form $\boldsymbol{f}\ (\boldsymbol{f}^0\ S_0) \ldots (\boldsymbol{f}^n\ S_n)$, reducing it to the last argument of the spines $\mathsf{last}(S_0)$, provided all S_i are equal. When $n = 0$, we recover the equality $\uparrow\downarrow X = X$.

The main judgment is atomic object typed reduction $\Delta; \Gamma \vdash_\top F \Rightarrow \Delta'; F'; P$, which reads: in repository Δ and environment Γ, applicative term F is well-typed of type P, and the slicing of its value produces a new repository Δ'. F' is the *residual* of this operation: only constant applications $\mathsf{c}\ S$ are sliced, F' is the "tip" of F that was not sliced.

These typing rules use typed definitional equality between families $\Delta; \Gamma \vdash A_1 = A_2$ and spines $\Delta; \Gamma; A \vdash S_1 = S_2 : P$. The formal definition of these judgments is eluded here. It compares syntactically the respective values of its objects. Since it potentially triggers function evaluation (and thus typing verification), we carry the repository, the environment and the type of the compared objects.

To reduce the size of the stored environments and substitutions to their minimum in slices $\Gamma \vdash F : P$, we use an auxiliary operation:

Definition 5. Strengthening *takes an environment Γ and an object F well-typed in Γ to an environment Γ' containing only the free variables of F:* $\mathsf{stren}(\Gamma, F) = \{x : P \mid x \in \mathsf{FV}(F) \wedge \Gamma(x) = P\}$.

It is correct to strengthen Γ only w.r.t. F before storing F as a slice because of the following:

Lemma 1. *If $\Delta; \Gamma \vdash_L F : A \Rightarrow \Delta'; F'$, then $\Delta; \mathsf{stren}(\Gamma, F) \vdash_L F : A \Rightarrow \Delta'; F'$*

Definition 6. *Let $\mathcal{R} = (\Delta, X[\sigma])$. If $\Delta; \Gamma \vdash_\top F \Rightarrow \Delta'; X'[\sigma']; P$, then we define the* commit *operation $\mathsf{ci}_\mathcal{R}(F) = (\Delta', X[\sigma])$. We say that $\mathcal{R}$ is* well-constructed *if $\mathcal{R} = \mathsf{ci}_{\mathcal{R}'}(F)$ with either $\mathcal{R}' = (\cdot, X[\sigma])$ or $\mathcal{R}'$ well-constructed.*

Lemma 2. *If $\mathcal{R}$ well-constructed, then $\mathsf{co}(\mathcal{R})$ is defined and is a value.*

We can already state a conservativity result stating that our algorithm accepts exactly the well-typed values of LF:

Theorem 1. *If F is a value, and $\mathcal{R}$ is well-constructed then $\mathsf{ci}_\mathcal{R}(F) = \mathcal{R}'$ iff $\Gamma \vdash_{\mathsf{LF}} \mathsf{co}_{\mathcal{R}'}(F) : A$*

$\boxed{\Delta; \Gamma \vdash_L M : A \Rightarrow \Delta'; M'}$ Canonical object

$$\textsc{MLam}\ \frac{\Delta; \Gamma, x : A \vdash_{\eta(L)} M : A' \Rightarrow \Delta'; M'}{\Delta; \Gamma \vdash_L \lambda x.\ M : \Pi x : A.\ A' \Rightarrow \Delta'; \lambda x.\ M'} \qquad \begin{array}{l}\eta(\top) = \top \\ \eta(\bot) = \bot \\ \eta(\boldsymbol{f}) = \bot\end{array}$$

$$\textsc{MAtom}\ \frac{\Delta; \Gamma \vdash_L F \Rightarrow \Delta'; F'; P' \qquad \Delta'; \Gamma \vdash P = P'}{\Delta; \Gamma \vdash_L F : P \Rightarrow \Delta'; F'}$$

$\boxed{\Delta; \Gamma \vdash_L F \Rightarrow \Delta'; F'; P}$ Atomic object

$$\textsc{FAppVar}\ \frac{\Delta; \Gamma; A \vdash_{\eta(L)} S \Rightarrow \Delta'; S'; P}{\Delta; \Gamma \vdash_L x\ S \Rightarrow \Delta'; x\ S'; P} \quad (x : A) \in \Gamma$$

$$\textsc{FAppConst}\ \frac{\Delta; \Gamma; A \vdash_{\eta(L)} S \Rightarrow \Delta'; S'; P}{\Delta; \Gamma \vdash_L \mathsf{c}\ S \Rightarrow \Delta'; \mathsf{c}\ S'; P} \quad \begin{array}{l}(c : A) \in \Sigma \\ L \neq \top\end{array}$$

$$\textsc{FAppConstSlice}\ \frac{\Delta; \Gamma; A \vdash_\top S \Rightarrow \Delta'; S'; P \qquad \mathsf{stren}(\Gamma, \mathsf{c}\ S') = \Gamma'}{\Delta; \Gamma \vdash_\top \mathsf{c}\ S \Rightarrow \Delta'[X \mapsto (\Gamma' \vdash \mathsf{c}\ S' : P)]; X[\mathsf{id}(\Gamma')]; P} \quad \begin{array}{l}(c : A) \in \Sigma \\ (X \text{ fresh in } \Delta')\end{array}$$

$$\textsc{FAppEval}\ \frac{\begin{array}{c}\Delta; \Gamma; A \vdash_{L'} S \Rightarrow \Delta'; S'; P \\ \Delta'; \Gamma \vdash T \star S' \Rightarrow \Delta''; F \qquad \Delta''; \Gamma \vdash_L F : P \Rightarrow \Delta'''; F'\end{array}}{\Delta; \Gamma \vdash_L \boldsymbol{f}\ S \Rightarrow \Delta'''; F'; P} \quad \begin{array}{l}(\boldsymbol{f} : A = T) \in \Sigma \\ S' \neq (\boldsymbol{f}^0\ S_0)\ \ldots\ (\boldsymbol{f}^n\ S_n) \\ (L = \boldsymbol{g} \wedge L' = \bot) \vee (L = \top \wedge L' = \boldsymbol{f})\end{array}$$

$$\textsc{FAppEvalInv}\ \frac{\begin{array}{c}\Delta; \Gamma; A \vdash_f S \Rightarrow \Delta'; (\boldsymbol{f}^0\ S_0)\ \ldots\ (\boldsymbol{f}^n\ S_n); P \\ \text{for all } i,j,\ \ (\Delta; \Gamma; A \vdash S_i = S_j : P)\end{array}}{\Delta; \Gamma \vdash_L \boldsymbol{f}\ S \Rightarrow \Delta'; \mathsf{last}(S_0); P} \quad (L = \top \vee L = \boldsymbol{g})$$

$$\textsc{FAppNoEval}\ \frac{\Delta; \Gamma; A \vdash_\bot S \Rightarrow \Delta'; S'; P}{\Delta; \Gamma \vdash_\bot \boldsymbol{f}\ S \Rightarrow \Delta'; \boldsymbol{f}\ S'; P} \quad (\boldsymbol{f} : A = T \in \Sigma)$$

$$\textsc{FMeta}\ \frac{\Delta(X) = (\Gamma' \vdash F : P) \qquad \Delta; \Gamma \vdash_{\eta(L)} \sigma : \Gamma' \Rightarrow \Delta'; \sigma'}{\Delta; \Gamma \vdash_L X[\sigma] \Rightarrow \Delta'; X[\sigma']; P[\sigma']}$$

Fig. 3. Typed evaluation for objects (1)

$\boxed{\Delta; \Gamma; A \vdash_L S \Rightarrow \Delta'; S'; P}$ Spine

$$\frac{\Delta; \Gamma \vdash_L M : A \Rightarrow \Delta'; M' \qquad \Delta'; \Gamma; A'[x/M'] \vdash_L S \Rightarrow \Delta''; S'; P}{\Delta; \Gamma; \Pi x : A.\ A' \vdash_L M\ S \Rightarrow \Delta''; M'\ S'; P}\ \textsc{SCons} \qquad \frac{}{\Delta; \Gamma; P \vdash_L \cdot \Rightarrow \Delta; \cdot; P}\ \textsc{SNil}$$

$\boxed{\Delta; \Gamma \vdash_L \sigma : \Gamma' \Rightarrow \Delta'; \sigma'}$ Substitution

$$\frac{\Delta; \Gamma \vdash_L \sigma : \Gamma' \Rightarrow \Delta'; \sigma' \qquad \Delta'; \Gamma \vdash_{\eta(L)} M : A[\sigma'] \Rightarrow \Delta''; M'}{\Delta; \Gamma \vdash_L (\sigma, x/M) : (\Gamma', x : A') \Rightarrow \Delta''; (\sigma', x/M')}\ \sigma\textsc{Cons} \qquad \frac{}{\Delta; \Gamma \vdash_L \cdot : \cdot \Rightarrow \Delta; \cdot}\ \sigma\textsc{Nil}$$

Fig. 4. Typed evaluation for objects (2)

Rule FAppEval calls the evaluation of a function, written as a CL term, on a spine. The rules of Fig. 5 follow the call-by-name strategy, so that when destructed by a case analysis an atom is only head-reduced. We finish by a soundness result:

Theorem 2. *If $\mathcal{R}$ well-constructed, and $ci_{\mathcal{R}}(F)$ succeeds with $\mathcal{R}'$, then $co(\mathcal{R}')$ is a closed value, well-typed w.r.t.* LF.

3 Related work

Symbolic manipulation of proofs Several attempts to design a functional programming language are being developed to compute over valid proof derivations [15–17]. Two essential design aspects of these tools are the choice of the representation for proof trees and the choice of the meta-language to manipulate them. Like them, we commit ourselves to Higher-Order Abstract Syntax (HOAS), and more specifically to a language of the LF family. This recent work tackles the ambitious challenge of deciding statically if a proof-generating function will always produce valid proof terms. Unlike them, we follow a contract-based approach [7] to guarantee the validity of the generated proofs. Thus, our technique is similar to LF-based proof-carrying code [2, 18], and to the LCF architecture: an untrusted proof generator is composed with a small, trusted proof checker. A noticeable difference is the fact that, in our setting, the checking of proofs is interleaved with their generation, allowing earlier adn easier bug catching during their development. On one hand, these approaches are more satisfying and effective because of the static guarantees. On the other hand, they are based on more sophisticated type systems, which augments the trusted base, and put more constraints on the way programs are written. When programming type checkers as well as in other examples, we encounter the problem of linking a variable x to a typing hypothesis $\vdash x : A$ to retrieve its type. In [15], a machinery of environment-side Σ-types is proposed. We replace it by the use of inverse function which seems more general, but puts more burden on the reduction strategy.

$\boxed{\Delta; \Gamma \vdash T \star S \Rightarrow \Delta'; F}$ Term evaluation

$$\textsc{TLam}\ \frac{\Delta; \Gamma \vdash T[x/M] \star S \Rightarrow \Delta'; F}{\Delta; \Gamma \vdash \lambda x.\ T \star M\ S \Rightarrow \Delta'; F} \qquad \textsc{TAtom}\ \frac{\Delta; \Gamma \vdash U \Rightarrow \Delta'; F}{\Delta; \Gamma \vdash U \star \cdot \Rightarrow \Delta'; F}$$

$\boxed{\Delta; \Gamma \vdash U \Rightarrow \Delta'; F}$ Atomic term evaluation

$$\textsc{UAtom}\ \frac{\Delta; \Gamma \vdash_{\boldsymbol{f}} F \Rightarrow \Delta'; F'; P}{\Delta; \Gamma \vdash F \Rightarrow \Delta'; F'}\ \begin{array}{l}\boldsymbol{f} \text{ fresh} \\ F' \neq X[\sigma]\end{array} \qquad \textsc{UMeta}\ \frac{\Delta; \Gamma \vdash_{\boldsymbol{f}} F \Rightarrow \Delta'; X[\sigma]; P}{\Delta; \Gamma \vdash F \Rightarrow \Delta'; \Delta'(X[\sigma])}\ (\boldsymbol{f} \text{ fresh})$$

$$\textsc{UCase}\ \frac{\Delta; \Gamma @ \Gamma' \vdash U \Rightarrow \Delta'; H_i\ M_1\ \ldots\ M_{m_i} \qquad \Delta'; \Gamma \vdash U_i[x_1/M_1 \ldots\ x_{m_i}/M_{m_i}] \Rightarrow \Delta''; F}{\Delta; \Gamma \vdash \textbf{case}\ U\ \textbf{in}\ \Gamma'\ \textbf{of}\ (H_1\ x_{11}\ \ldots\ x_{1m_1} \to U_1) \mid \ldots \mid (H_n\ x_{n1}\ \ldots\ x_{nm_n} \to U_n) \Rightarrow \Delta''; F}$$

Fig. 5. Evaluation of CL terms

Incremental computation Memoization is a widely used technique to reuse previously computed output [3]. The slicing of our values implements the same idea: naming intermediate results of computation (derivations), and reuse of these computation results by metavariables. Whereas memoization embeds the recognition of a previously seen input, we delegate this to the user (or a front-end) who is responsible for providing the delta: these deltas are more involved to infer in the higher-order case, and may invoke reasons that are beyond the scope of a basic memoized type-checker to justify the reuse of a specific typing derivation. For instance, if term $\lambda x : A.\ M$ has been assigned the type $A \to B$ leading to a typing derivation $\mathcal{D}$, one can specialize its type to $A' \to B$ by applying the subtyping rule directly on $\mathcal{D}$ and providing a proof that $A' \to B \leq A \to B$.

There is more to incrementality than just memoization. Adaptive functional programming techniques [4, 5] includes memoization but also, thanks to dependency tracking, allows to recompute the output only for changed parts without even considering unchanged parts. We do have an explicit representation of changes (the deltas), but they still must describe the path down to the changed part.

Conclusion and future work

We have presented a lightweight framework to program incremental certifying type-checker. A prototype implementation is available online[3] as well as examples of type-checkers for several variants of the λ-calculus. This prototype comes as an OCaml library with a CamlP4 syntax extension, offering the *commit* and *checkout*

[3] `http://www.cs.unibo.it/~puech/`

operations on repositories, given an LF signature and OCaml implementations for the defined functions.

This system is meant to serve as a kernel for checking deltas inferred by a front-end whose development is still to be done. This piece of software will be responsible for comparing several versions of the same source code to guess which modifications have been applied to it.

References

1. Leroy, X.: Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In: 33rd symposium Principles of Programming Languages, ACM Press (2006) 42–54
2. Necula, G.: Proof-carrying code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM (1997) 106–119
3. Pugh, W., Teitelbaum, T.: Incremental computation via function caching. In: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM (1989) 315–328
4. Acar, U.A., Blelloch, G.E., Harper, R.: Adaptive functional programming. ACM Trans. Program. Lang. Syst. **28**(6) (November 2006) 990–1034
5. Carlsson, M.: Monads for incremental computing. In: ICFP. (2002) 26–35
6. Nanevski, A., Pfenning, F., Pientka, B.: Contextual modal type theory. ACM Transactions on Computational Logic (TOCL) **9**(3) (2008) 23
7. Wadler, P., Findler, R.: Well-typed programs can't be blamed. Programming Languages and Systems (2009) 1–16
8. Boespflug, M.: Conception d'un noyau de vérification de preuves pour le lambda-Pi-calcul modulo. PhD thesis, École Polytechnique, Palaiseau (January 2011)
9. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the Association for Computing Machinery **40**(1) (1993) 143–184
10. Pfenning, F., Elliot, C.: Higher-order abstract syntax. ACM SIGPLAN Notices **23**(7) (1988) 199–208
11. Pfenning, F., Simmons, R.J.: Term representation. Notes of the LF seminar, Meeting 1 (2007)
12. Harper, R., Licata, D.R.: Mechanizing metatheory in a logical framework. Journal of Functional Programming (2007)
13. Herbelin, H.: A λ-calculus structure isomorphic to gentzen-style sequent calculus structure. In: Computer Science Logic, Springer (1995) 61–75
14. Pierce, B., Turner, D.: Local type inference. ACM Transactions on Programming Languages and Systems (TOPLAS) **22**(1) (2000) 1–44
15. Cave, A., Pientka, B.: Programming with binders and indexed data-types. SIGPLAN Not. **47**(1) (January 2012) 413–424
16. Stampoulis, A., Shao, Z.: Static and user-extensible proof checking. In: POPL. (2012) 273–284
17. Poswolsky, A., Schürmann, C.: Practical programming with higher-order encodings and dependent types. In: Proceedings of the Theory and practice of software, 17th European conference on Programming languages and systems. ESOP'08/ETAPS'08, Berlin, Heidelberg, Springer-Verlag (2008) 93–107
18. Appel, A., Felten, E.: Proof-carrying authentication. In: Proceedings of the 6th ACM Conference on Computer and Communications Security, ACM (1999) 52–62