Interaction Trees and Verified Compilation

Extended Abstract

Paolo Torrini, INRIA Sophia-Antipolis, France
Benjamin Gregoire, INRIA Sophia-Antipolis, France
Paolo.Torrini@inria.fr
Benjamin.Gregoire@inria.fr

Abstract

We have used Interaction Trees (ITrees) to formalize and verify the front-end of the Jasmin compiler, relying on an integration of coinductive denotational semantics with Relational Hoare Logic. A crucial element in our approach is the definition of an appropriate relation to compare programs, which makes it possible to simplify the reasoning about undefined behaviour.

ITrees [11] can be used to reason denotationally about possibly diverging programs in an executable, modular way, thus providing an attractive alternative to the operational semantics used in major verified compilation projects such as CompCert [7] and CakeML [6]. The coinductive datatype itree $E\ V$, parameterized by a type family of events E and a final result type V, represents computations as trees with possibly infinite branches. A leaf Ret v is termination with value $v : V$. A unary node Tau t is a silent transition step which continues with t : itree $E\ V$. A multi-branching node Vis $A\ e\ k$ is a computation that executes the event $e : E\ A$ and feeds the answer to the continuation $k : A \longrightarrow$ itree $E\ V$. Being monad themselves, ITrees can be executed by layering monadic interpreters, each associated with a handler for a subfamily of events. Generalized recursion can be represented as corecursive interpretation of call events relying on a monadic iterator. Failures can be represented as events that return in the null type, to be interpreted using an error monad transformer. ITrees facilitate coinductive reasoning through a rich equational theory [1]. Modularity is supported by disjoint unions of event families (e.g. $E_1 +' E_2$) and interpretation layers, possibly associated with monad transformers.

Jasmin [2, 3] is a comparatively low-level language intended for security applications, designed to give the programmer explicit control on the generation of assembly code. It includes operators and instructions on memory and registers, conditional statements, while loops, internal and external function calls. It is formalized in Rocq with a fully verified compiler based on a big-step semantics for terminating programs and a concrete memory model. In order to lift the termination restriction, we defined a new semantics with ITrees and used it to verify so far the compiler front-end, based on a language that remains syntactically the same while being semantically interpreted differently after each pass (more than twenty ones, including constant propagation, dead code elimination, inlining and stack allocation).

Reasoning about compilation relies on the definition of behavioural equivalence between source and target, which can be used either for forward or backward reasoning. Some subtlety is involved in dealing with abnormal termination. Compilation is meant to preserve normal behaviour, but in general the compiler may not worry about what happens when the source throws a run-time error. Indeed, compiler correctness proofs are often restricted to *safe* executions, i.e. those that do not terminate abnormally. Verified compilation is meant to preserve safety as part of behaviour preservation, and therefore it suffices to assume that the initial source is safe. However, the safety restriction makes proofs less general and can complicate compositionality. In order to lift this restriction, in our verification we use a notion of program equivalence which makes it possible to express that the compiled code behaves like the source only up to events called *cutoffs*. Cutoff events can be used to represent source errors which are meant to be matched with any behaviour in the target, as causal instances of undefined behaviour (UB). Notice that unlike C and LLVM, Jasmin has a deterministic semantics, but interpreting failure as UB helps to make the semantics simpler.[1]

The ITree library [1] provides generalizations of bisimilarity, both strong ($\cong_R$) and weak ($\approx_R$, called *equivalence up-to-tau*), parameterized by a relation R between result types, allowing for trees which are heterogeneous in the returned value types to be related, provided they use the same events. Instantiating R with equality gives us the usual strong and weak bisimilarity. The fact that, regardless of R, related events must be equal, makes it hard to directly relate function calls across a translation while keeping into account differences in the types of call arguments and results. A way around this problem, showcased in [4, 11–13] consists in layering interpretations, separating a common ground of ITrees based on the same events which can be compared directly, from higher-level layers which need to be compared in terms of more complex notions of refinement, also using predicates on ITrees [4, 13] to account for non-determinism (a problem we do not need to deal with). A more intensional way to relate programs is given by the event-heterogeneous relation introduced in [10], used in [4, 9] and here denoted $\approx^e_R$, as a generalization of $\approx_R$ which is parameterized by two additional relations: a precondition specifying which events

[1] Unlike [4] we do not consider time-travelling UB.

can be related, inclusive of their inputs, and a postcondition which is expected to hold between the outputs of event execution. $\approx^e_R$ makes it possible to capture structural similarity more directly, yet it does not let us interpret source errors as UB, as it does not allow relating a terminating program with a silently non-terminating one. While needed with normal termination, this restriction is unnecessary when dealing with an abnormal one.

We can answer this problem by a generalization of $\approx^e_R$, denoted $\approx^u_R$, which we call *equivalence up-to-cutoff*. As much as $\approx_R$ and $\approx^e_R$, $\approx^u_R$ is coinductively defined as the greatest fixed point of an inductive relation. Its definition is illustrated in Figure 1, where a single bar means that the hypothesis is used inductively, while a double bar means that it is used coinductively (or simply non-inductively). $\approx^u_R$ can relate wholly heterogeneous trees t_1 : itree E_1 V_1 and t_2 : itree E_2 V_2. The rules Ret, Tau, Tau$_l$, Tau$_r$, and Vis are those of $\approx^e_R$. Ret allows relating return values by R. In the Vis rule, Φ acts as relational precondition and Ψ as postcondition for the event handlers used to interpret the events. As in [10, 11] the fact that rules Tau$_l$ and Tau$_r$ rely on their hypothesis inductively ensures that the related trees can only differ up to a finite number of silent steps. Specifically to $\approx^u_R$, there are as additional parameters two Boolean predicates C^l on E_1 events and C^r on E_2 ones, deciding which are the cutoffs on each side, and two additional rules Cut$_l$ and Cut$_r$, which allow for a cutoff to be related with any tree.[2] $\approx^u_R$ supports setoid rewriting, enjoys structural properties, monotonicity, transitivity, and it is preserved by interpretation (Intr) for related handlers (h_1, h_2):

$$t_1 \approx^u_R t_2 \to (\forall e_1 e_2.\ h_1\ e_1 \approx^u_R h_2\ e_2) \to \mathsf{Intr}_{h_1}\ t_1 \approx^u_R \mathsf{Intr}_{h_2}\ t_2$$

Our intended semantics for Jasmin is to relate a source failure to any behaviour of the target program beyond that point. Using $\approx^u_R$, it suffices to declare source failures as cutoff events. We write $\approx^u_{R\ \Phi\ \Psi\ C^l\ C^r}$ for the fully parameterized relation. We also write $\lceil p_i \rceil_{s_i}$, depending on state $s_i : S_i$, for the semantic interpretation of program $p_i : \mathcal{L}$ (after the i-th pass) as an ITree of type itree $(F +' G)$ S_i where F is the type family of failure events. Further writing R_i for the relation between states across the pass, Φ_i for the precondition that holds between events and Ψ_i for the corresponding postcondition, C_F for the predicate that sets F events as the only cutoffs and $C_\emptyset$ as the predicate that sets no cutoffs, the verification of a two-pass compilation (where related events may differ even if they belong to the same family) can be built transitively out of two components:

$$\frac{\vdash_\Gamma \lceil p_0 \rceil_{s_o} \approx^u_{R_1\Phi_1\Psi_1 C_F C_\emptyset} \lceil p_1 \rceil_{s_1} \quad \vdash_\Gamma \lceil p_1 \rceil_{s_1} \approx^u_{R_2\Phi_2\Psi_2 C_F C_\emptyset} \lceil p_2 \rceil_{s_2}}{\vdash_\Gamma \lceil p_0 \rceil_{s_0} \approx^u_{(R_1\circ R_2)(\Phi_1\circ\Phi_2)(\Psi_1\circ\Psi_2) C_F C_\emptyset} \lceil p_2 \rceil_{s_2}}$$

This principle holds regardless of the events in G (which are still uninterpreted). Nonetheless, our current verification

[2] A more specific form of this relation was already used in [4].

$$\frac{r_1\ R\ r_2}{\mathsf{Ret}(r_1) \overset{u}{\approx} \mathsf{Ret}(r_2)}\ \textsc{Ret} \qquad \frac{t_1 \overset{u}{\approx} t_2}{\mathsf{Tau}(t_1) \overset{u}{\approx} \mathsf{Tau}(t_2)}\ \textsc{Tau}$$

$$\frac{C^l(e_1)}{\mathsf{Vis}(e_1, k_1) \overset{u}{\approx} t_2}\ \textsc{Cut}_l \qquad \frac{C^r(e_2)}{t_1 \overset{u}{\approx} \mathsf{Vis}(e_2, k_2)}\ \textsc{Cut}_r$$

$$\frac{t_1 \overset{u}{\approx} t_2}{\mathsf{Tau}(t_1) \overset{u}{\approx} t_2}\ \textsc{Tau}_l \qquad \frac{t_1 \overset{u}{\approx} t_2}{t_1 \overset{u}{\approx} \mathsf{Tau}(t_2)}\ \textsc{Tau}_r$$

$$\frac{e_1\ \Phi\ e_2 \qquad \forall v_1\ v_2.\ (e_1, v_1)\ \Psi\ (e_2, v_2) \implies k_1(v_1) \overset{u}{\approx} k_2(v_2)}{\mathsf{Vis}(e_1, k_1) \overset{u}{\approx} \mathsf{Vis}(e_2, k_2)}\ \textsc{Vis}$$

Figure 1. Equivalence up-to-cutoff ($\approx^u$), parameterized by R, Φ, Ψ, C^l and C^r.

makes limited use of event layering, as we leaned significantly on reuse of existing low-level proofs. Indeed, on top of $\approx^u_R$, we relied on a form of Relational Hoare Logic, originally introduced in [5] which we specialized to ITrees, allowing us to obtain proofs that are comparatively similar to those originally made with the inductive big-step semantics. With the notable exception of inlining, it was possible to make our proofs relying on coinductive lemmas that were either provided by [1] or did not pose particular challenges.

Our compiler correctness statements, which we basically proved by induction on source p for the single passes, then gluing the pieces together by transitivity, have general form:

$$\forall p\ s\ s',\ s\ R\ s' \to \lceil p \rceil_s \approx^u_{R\Phi\Psi C_F C_\emptyset} \lceil p' \rceil_{s'}$$

While our proofs follow in the footsteps of forward simulation given the determinism of the target, the double-sided character of $\approx^u_R$ can equally support backward reasoning, allowing us to backward-match cutoff-set target errors with any further source behaviour. We proved a generalized form of transitivity:

$$\frac{\begin{array}{ll}\vdash_\Gamma \forall e,\ \neg(C^r_1\ e \wedge C^l_2\ e) & \vdash_\Gamma \forall e e',\ e\ \Phi_1\ e' \wedge C^l_2\ e' \to C^l_1\ e \\ & \vdash_\Gamma \forall e e',\ e\ \Phi_2\ e' \wedge C^r_1\ e \to C^r_2\ e' \\ \vdash_\Gamma t \approx^u_{R_1\Phi_1\Psi_1 C^l_1 C^r_1} t' & \vdash_\Gamma t' \approx^u_{R_2\Phi_2\Psi_2 C^l_2 C^r_2} t''\end{array}}{\vdash_\Gamma t \approx^u_{(R_1\circ R_2)(\Phi_1\circ\Phi_2)(\Psi_1\circ\Psi_2) C^l_1 C^r_2} t''}$$

As far as UB-related non-determinism goes, this property could help us in combining together backward and forward reasoning, possibly relaxing safety preservation to integrate with safety analysis on targets.

As future work, we want to cover the Jasmin back-end (which includes linearization). $\approx^u_R$ can also be used to relate trees up to cutoff-determined prefixes, possibly making step-indexing easier, and indeed we are interested in the comparison with step-indexing and fuel-based inductive semantics [8], as well as in safety analysis.

References

[1] 2025. *Interaction Trees, GitHub Repository.* https://github.com/DeepSpec/InteractionTrees

[2] 2025. *Jasmin, GitHub Repository.* https://github.com/jasmin-lang/jasmin

[3] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. 2017. Jasmin: High-Assurance and High-Speed Cryptography. In *Proc. ACM Computer and Communication Security (CCS 2017)*, Bhavani Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu (Eds.). ACM, 1807–1823. https://doi.org/10.1145/3133956.3134078

[4] Calvin Beck, Irene Yoon, Hanxi Chen, Yannick Zakowski, and Steve Zdancewic. 2024. A Two-Phase Infinite/Finite Low-Level Memory Model: Reconciling Integer-Pointer Casts, Finite Space, and undef at the LLVM IR Level of Abstraction. *Proc. ACM Program. Lang.* 8, ICFP (2024), 789–817. https://doi.org/10.1145/3674652

[5] Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, Neil D. Jones and Xavier Leroy (Eds.). ACM, 14–25. https://doi.org/10.1145/964001.964003

[6] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 179–192. https://doi.org/10.1145/2535838.2535841

[7] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (2009), 107–115.

[8] Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-Step Semantics. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*. Springer-Verlag, Berlin, Heidelberg, 589–615. https://doi.org/10.1007/978-3-662-49498-1_23

[9] Lucas Silver, Eddy Westbrook, Matthew Yacavone, and Ryan Scott. 2023. Interaction Tree Specifications: A Framework for Specifying Recursive, Effectful Computations That Supports Auto-Active Verification. In *Proc. ECOOP 2023 (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 30:1–30:26. https://doi.org/10.4230/LIPIcs.ECOOP.2023.30

[10] Lucas Silver and Steve Zdancewic. 2021. Dijkstra monads forever: termination-sensitive specifications for interaction trees. *Proc. ACM Program. Lang.* 5, POPL (2021), 28 pages. https://doi.org/10.1145/3434307

[11] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. https://doi.org/10.1145/3371119

[12] Irene Yoon, Yannick Zakowski, and Steve Zdancewic. 2022. Formal reasoning about layered monadic interpreters. *Proc. ACM Program. Lang.* 6, ICFP (2022), 254–282. https://doi.org/10.1145/3547630

[13] Yannick Zakowski, Calvin Beck, Irene Yoon, Ilia Zaichuk, Vadim Zaliva, and Steve Zdancewic. 2021. Modular, compositional, and executable formal semantics for LLVM IR. *Proc. ACM Program. Lang.* 5, ICFP (2021), 1–30. https://doi.org/10.1145/3473572