Deductive Verification via Ghost Debugging

Martin Clochard, Andrei Paskevich, Claude Marché

Deductive Verification via Ghost Debugging

Martin Clochard, Andrei Paskevich , Claude Marché

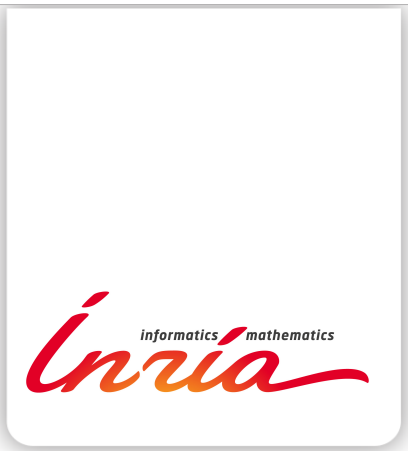
RESEARCH REPORT

N° 9219

Octobre 2018

Project-Team Toccata

ISSN 0249-6399 ISRN INRIA/RR--9219--FR+ENG



Deductive Verification via Ghost Debugging

Martin Clochard*, Andrei Paskevich† ‡, Claude Marché‡†

Project-Team Toccata

Research Report n° 9219 — Octobre 2018 — 29 pages

Abstract: We present a verification approach based on auxiliary programs, which we call *ghost debuggers*. This approach leads to notable verification gains when the structure of the program does not match the execution structure, notably when that latter structure is recursive. We present a theoretical foundation of our approach over a flavor of transfinite games, which let us specify and prove fine-grained properties about infinite behaviors of programs. By making use of the game structure, we also show that our approach can be applied to relational properties like simulation between programs. Our approach is backed by a mechanized development in the Why3 tool, which formally proves sound the Hoare-style logic backbone of our approach.

Key-words: Deductive Verification, Infinite executions, Simulation, Games

Work partly supported by the Joint Laboratory ProofInUse (ANR-13-LAB3-0007, `http://www.spark-2014.org/proofinuse`) of the French national research organization

* ETH Zürich
† LRI, Université Paris-Sud & CNRS, F-91405 Orsay
‡ Inria, Université Paris-Saclay, F-91120 Palaiseau

RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Vérification déductive via débogage fantôme

Résumé : Nous présentons une nouvelle approche de vérification de programmes que nous appelons *Débogage Fantôme*. Cette approche apporte des gains notables lorsque la structure syntaxique du code source du programme à vérifier ne correspond pas à la structure de son exécution, notamment lorsque cette dernière est récursive. Nous présentons un fondement théorique de notre approche sur une variante de jeux transfinis, qui nous permet de spécifier et prouver des propriétés fines sur les comportements infinis des programmes. En utilisant la structure du jeu, nous montrons également que notre approche peut être appliquée à des propriétés relationnelles comme la simulation entre les programmes. Notre approche est formalisée par un développement dans l'outil Why3, qui nous permet de valider mécaniquement la correction de la logique à la Hoare qui sous-tend notre approche.

Mots-clés : Vérification déductive, exécutions infinies, simulation, jeux

Contents

1 Introduction

We propose a method for deductive program verification that relies on external auxiliary programs to make explicit the underlying algorithmic structure of the target program and supports fine-grained verification of non-terminating behaviors. Our method is based on transfer of correctness properties from auxiliary programs to the verification target. The core of our approach, presented in Section 2, amounts to writing an auxiliary program that controls the execution of the verification target, in a way that ensures the correctness transfer. To achieve this, we only need the small-step operational semantics of the programming language. We then use a weakest precondition calculus to prove the auxiliary program. We observe that the additional structure of the auxiliary program, notably recursive structure, can lead to notable verification gains. Moreover, since the proof technique only requires structure for the auxiliary program, it can be applied to unstructured target code.

The theoretical foundation of our approach uses the framework of games, as presented in Section 3. This let us generalize our approach with respect to the interpretation of non-determinism. A universal interpretation of non-determinism, commonly termed *demonic*, is used to prove program correctness for every possible behavior. On the other hand, an existential, *angelic*, interpretation is used to prove the existence of specific behaviors, or the existence of valid implementation choices. The use of games to obtain a Hoare logic or a weakest pre-condition calculus for programs using both kinds of non-determinism is not new, and notably appears in [2] and [6]. We show in Section 4 that our approach can be used to prove at least three kinds of properties: program correctness, existence of a given behavior for a given program, and, finally, simulation between two programs. Note that the third one is a relational property, and can be leveraged for proving compiler correctness. We expect that this framework can also be used to prove other relational properties, like determinism.

A consequence of our game framework is that we can smoothly support non-terminating recursion within auxiliary programs, and continuations to represent control structures like `break` or `continue`. Moreover, we make our framework applicable to fine-grained verification of infinite behavior by using two techniques. The first one is to use the least upper bound to represent infinite behavior, typically via the trace of all observable events. We use this technique to distinguish distinct infinite behaviors. The second one is to support a divergence handling mechanism in our auxiliary programs, which catches divergence as if it was an exception. This mechanism works even if the diverging auxiliary program represents a finite run of the underlying program, and as a bonus extends to both transfinite auxiliary recursive programs and transfinite plays.

Not all the elements that we present here are new. We mentioned before that the framework of games is a classical method to represent dual non-determinism, and combines well with Hoare logic. Moreover, if we view auxiliary program constructions as proof rules, then the approaches for machine-code verification in the style of [8] match our approach for an auxiliary `While` programming language augmented with well-founded recursion. Our own contributions are summarized as follows:

- We show through examples that the method of verification based on auxiliary programs can be used as a general method to reduce verification effort, which includes the case of a structured programming language. This method notably applies well to the proof of Schorr-Waite algorithm [10]. In particular, we argue that this verification style should not be limited to the niche of assembly-like languages.
- We provide a game-based weakest pre-condition calculus for auxiliary programs that supports both arbitrary recursive functions and fine-grained specification and proof of non-terminating behaviors. This weakest pre-condition calculus applies to transfinite auxiliary programs, and does not require a correlation between divergence of the auxiliary program and of the verification target.
- We show that our approach applies not only to the usual correctness and behavior existence properties, but also to relational properties like simulation. As we do not require any correlation between

the structure of the target code and the auxiliary code, we can apply this method for arbitrary simulation results.

- In order to provide stronger guarantees about our results, we mechanized the Hoare logic underlying our language for auxiliary programs using the Why3 tool. The development is available online at URL `http://toccata.lri.fr/gallery/hoare_logic_and_games.en.html`

2 Informal Presentation of the Approach

2.1 Toy Example: McCarthy's 91 Function

We consider a toy example, based on the famous "91 function" of McCarthy [7]. It is defined by the recursive equation

$$f(n) = \texttt{if } n \leq 100 \texttt{ then } f(f(n+11)) \texttt{ else } n - 10 \tag{1}$$

Although the recursion scheme is tricky, it can be shown that the recursive evaluation of $f(n)$ terminates for every integer n, and more precisely that $f(n) = spec(n)$ where

$$spec(n) = \texttt{if } n \leq 100 \texttt{ then } 91 \texttt{ else } n - 10 \tag{2}$$

The correctness of the program code (1) with respect to the post-condition (2) can be proved in a fraction of a second by automated program verifiers existing nowadays (Dafny, F*, KeY, Why3, etc.) using any modern SMT solver as a backend (see, for example, `http://toccata.lri.fr/gallery/mccarthy.en.html`). The termination of $f(n)$ is just as easy to prove using the appropriate *variant* (i.e., decreasing measure) $101 - n$.

The code below is an iterative variant of (1), obtained by some manual de-recursification, using an extra variable e to represent the number of pending recursive calls.

```
e = 1; r = n;
while (1) {
  if (r > 100) { r = r - 10; e = e - 1;
                 if (e = 0) break; }
  else { r = r + 11; e = e + 1; }
}
```

Formally proving that this iterative program is terminating and computes $spec(n)$ (the result being in `r`) is significantly harder than on the recursive code: it requires to discover a more complex variant to prove its termination, and it also requires to discover a non-trivial loop invariant.

The approach we present in this paper aims at proving this kind of low-level imperative code without requiring more complex annotations than a more high-level and less efficient code. It amounts to re-interpret the iterative program into a more abstract program, on which we can reason more easily: in this case as simply as the original recursive definition (1). The basic idea is first to make more explicit the operational semantics of the iterative code, by introducing a `CONT` procedure that behaves similarly to the "continue" action in a debugger. For this, we identify specific execution points to the code that correspond to break points:

```
P0: e = 1; r = n;
while (1) {
  P1: if (r > 100) { r = r - 10; e = e - 1; P2: if (e = 0) break; }
      else { r = r + 11;  e = e + 1; }
}
P3:
```

The break points must be chosen in such a way that the evolution of the symbolic state of the target program between two break points can be computed statically, e.g., via symbolic execution. In particular, break points must break loops.

The procedure `CONT` then operates on the variables e and r plus the special variable pc representing the current point of execution, $pc = i$ meaning that execution is at point $\mathtt{P}_i$. In a Hoare-style formal verification setting, the effect of `CONT` can be *specified* with the precondition $0 \leq pc < 3$ and the following set of post-conditions (where primed variables denote values in the post-state):

$$\begin{aligned} pc = 0 &\rightarrow pc' = 1 \wedge e' = 1 \wedge r' = n \\ pc = 1 \wedge r > 100 &\rightarrow pc' = 2 \wedge e' = e - 1 \wedge r' = r - 10 \\ pc = 1 \wedge r \leq 100 &\rightarrow pc' = 1 \wedge e' = e + 1 \wedge r' = r + 11 \\ pc = 2 \wedge e = 0 &\rightarrow pc' = 3 \wedge e' = e \wedge r' = r \\ pc = 2 \wedge e \neq 0 &\rightarrow pc' = 1 \wedge e' = e \wedge r' = r \end{aligned}$$

We can then define an auxiliary procedure whose only effect is to call `CONT` a certain number of times, while mimicking the original recursive structure of the code (1). This is what we call a *ghost debugger*.

```
let rec aux ()
  requires { pc = 1 ∧ e > 0 }
  variant { 101 − r }
  ensures { pc' = 2 ∧ r' = spec(r) ∧ e' = e − 1 }
= CONT(); (* pc is now 1 or 2 *)
  if pc = 1 then (aux(); CONT(); aux())

let main ()
  requires { pc = 0 }
  ensures { pc' = 3 ∧ r' = spec(n) }
= CONT(); aux(); CONT()
```

Notice the similarity of the structure of aux with the initial recursive equation (1). At the meta level, we note that the effect of execution of $main$ is necessarily equivalent to a sequence of calls to `CONT`. Since the contract of $main$ implies that it executes from point P_0 to point P_3, we can deduce that the post-condition $r' = spec(n)$ is necessarily valid on the low-level code. We call that a *transfer property*: whatever the structure of the ghost debugger program, since it only modifies the state of the low-level code through `CONT`, any property proved in the contract of the ghost program is also valid for the low-level code. The ghost debugger given above is proved nearly as-is by the Why3 program verifier, modulo a few syntactic details.

The approach we present hence amounts to add to arbitrary code a structure that is better compatible with the proof. Note that since we did not exploit the structure of the iterative program, the same method would have worked for the same program written in an unstructured language like assembly. Although we do not gain much for this particular simple example, we obtain simpler proofs for other iterative algorithm which mimics a recurrence. A typical example is the Schorr-Waite algorithm. We have proved it using the approach presented here [1], using the Why3 program verifier. Comparing with a direct proof made with the same tool [2], the proof is slightly shorter and use simpler concepts, notably because we do not have to represent the stack. Also, note that the example proved using our approach contains a lot of annotations for the `CONT()` procedure, whose generation can be automated, and generalize over record size. So a compact equivalent proof would be even smaller.

[1] `http://toccata.lri.fr/gallery/schorr_waite_via_recursion.en.html`
[2] `http://toccata.lri.fr/gallery/schorr_waite.en.html`

2.2 Example: Reasoning on Infinite Behaviors

We consider an example based on parsing of well-parenthesized words, also known as the Dyck language $\mathcal{D}$. Parsing such words can be done via a counter, like in the following C one-liner:

```
n = 0; while(n >= 0) if(getchar() = '(') n++; else n--;
```

If we restrict our attention to inputs made of infinite streams of parenthesis characters, then the above program has essentially three possible behaviors:

1. Terminating executions. In that case, the program terminates after reading $w')'$ from the input, where $w \in \mathcal{D}$.

2. Non-terminating executions for which the value of variable n tends to infinity. This happens when the program read the entirety of an infinite input made of blocks shaped as $w'('$, with $w \in \mathcal{D}$.

3. Non-terminating executions for which n always comes back to a finite value. This last scenario happens when the program returns the entirety of an infinite input partitioned in two: first a finite prefix made of $w'('$ blocks, with $w \in \mathcal{D}$, and an infinite suffix made of $'('w')'$ blocks.

In order to verify those properties, first and foremost we must have some mean to talk about the sequence of input operations performed by the program. We obtain this in plain Hoare logic by baking the whole sequence of input operations within an auxiliary variable $inputs$.

In fact, we can also use this value to describe the behavior of the program over non-terminating executions. We characterize the observable behavior of a non-terminating program through the value of the variable $inputs$ *after* an infinite execution takes place. We can do so because the variable $inputs$ only grows during execution, so it makes sense to consider its "limit" state after an infinite number of execution steps as the least upper bound of the intermediate values. We claim this is an adequate description of the observable behavior as this least upper bound is exactly the potentially infinite sequence of input operations performed by the program during that particular infinite execution. Hence a simple specification style for such properties is to add a case in post-conditions for non-terminating execution, referring to the $inputs$ variable. Nakata & Uustalu [9] follow a similar approach to develop a trace-based Hoare logic for potentially non-terminating While programs.

For the verification perspective, let us first focus on the terminating behaviors of the above program, and then add ingredients for non-terminating behaviors. Verifying the terminating behaviors is exactly checking the partial correctness. We can readily apply the ghost debugging method to that proof. As before, we first identify specific execution points in the original program:

```
P0: n = 0; while(n >= 0) P1:if(getchar() = '(') n++; else n--; P2:
```

The procedure `CONT` for that annotated program can be specified by the pre-condition $0 \leq pc < 2$ and the post-condition

$$\begin{array}{rl} pc = 0 & \rightarrow\ pc' = 1 \wedge n' = 0 \wedge inputs' = inputs \\ pc = 1 \wedge n < 0 & \rightarrow\ pc' = 2 \wedge n' = n \wedge inputs' = inputs \\ pc = 1 \wedge n \geq 0 & \rightarrow\ pc' = 1 \wedge\ (\quad (n' = n + 1\ \wedge\ inputs' = inputs +' (') \\ & \qquad\qquad \vee (n' = n - 1\ \wedge\ inputs' = inputs +')') \quad) \end{array}$$

Input non-determinism is modeled by the disjunction in the third case. Note that alternatively, we could make `CONT` produce a result value to distinguish the distinct possibilities. We end the proof by verifying a ghost debugger, whose only effect is to call `CONT` for some amount of times. This annotated procedure, given in Figure 1, is verified as-is (modulo syntactic details) by Why3.

```
let rec aux ()
  requires { pc = 1 ∧ n ≥ 0 }
  ensures { pc' = 1 ∧ n' = n − 1 ∧ ∃w ∈ D. inputs' = inputs + w+')' }
= let n0, inputs0 = n, inputs in
  while true do
    invariant { pc = 1 ∧ n0 = n ∧ ∃w ∈ D. inputs = inputs0 + w }
    CONT(); (* read '(' or ')' *)
    if(n < n0) then return else aux()
  done

let main()
  requires { pc = 0 }
  ensures { pc' = 2 ∧ ∃w ∈ D. inputs' = inputs + w+')' }
= CONT(); aux(); CONT()
```

Figure 1: Ghost debugger for partial correctness of Dyck language parser

Since we are only proving partial correctness, we did not provide variants for recursive procedures and loops. As before, the transfer property ensures the partial correctness of our one-liner as a consequence. Note that the proof we propose use simple annotations (partition of the input in a few blocks), and corresponds directly to the representation of $\mathcal{D}$ by the following grammar:

$$\mathcal{D} ::= \epsilon \mid \mathcal{D}\mathcal{D} \mid (\mathcal{D})$$

We claim that this approach is more natural than a direct proof via a loop invariant. Indeed, a proof in that style would involves more complex arguments, based on a decomposition of the input in n blocks. Those blocks corresponds in fact to the stack of `aux()`, which must be made explicit in the annotation. Moreover, the auxiliary recursive function enjoys a nice property with respect to non-terminating behaviors: infinite recursion cause the second possible behavior, and infinite loop cause the third one. Hence we follow the verification mechanisms suggested by this disjunction. We control which non-terminating behaviors may occurs from the potentially non-terminating program constructions of the ghost debugger.

To this end, we treat non-termination as an exceptional behavior. In other words, we consider that non-terminating constructions raise an exception upon non-termination, containing a witness of the non-terminating run. We take that witness to be the relevant part of the non-termination trace. In the case of loops, this is the sequence of values taken by each variable at each iteration, while this is the sequence of values taken at each level of the infinite recursion stack for divergent recursive function execution. We can then analyze the witness content to determine the potential values of $inputs$, which is the least upper bound of the values of $inputs_i$ in the witness.

Note that to achieve our verification goal, we need to have information about the content of those witnesses. Indeed, if we have no information at all, we cannot do better than partial correctness statements. A first source of constraints over witnesses come from loop invariants and recursive function pre-conditions, which must hold for each elements of the sequence. Second, we know that $inputs$ only grows along the sequence, and that the value of $inputs$ is the least upper bound of values along that sequence. Third, we know that the first element of that sequence corresponds to the state before the loop or the call to recursively defined function. However, those constraints alone do not give us the desired block decomposition of the sequence. For that purpose, we equip the loop and the recursive function definition with a progression relation, which must hold between consecutive iterations or recursive calls. We can then exploit this relation between consecutive elements of the sequence of values. Note that the progression relation is the natural generalization of a decreasing measure for total correctness. We recover the

```
let rec aux()
  requires { pc = 1 ∧ n ≥ 0 }
  ensures  { pc' = 1 ∨ pc' = ∞ }
  ensures  { pc' = 1 → n' = n − 1 ∧ ∃w ∈ D. inputs' = inputs + w+')' }
  ensures  { pc' = ∞ → ∃N, (p_i)_{0≤i<N}, (w_j)_{j∈ℕ}.
    inputs' = inputs + Σ_{0≤i<N} (p_i +' (') + Σ_{0≤j} ('('+w_j+')') }
  progress { inputs with fun i i' => ∃w ∈ D. i' = i + w +' (' }
= let n_0, inputs_0 = n, inputs in
  try
    while true do
      invariant { pc = 1 ∧ n_0 = n ∧ ∃w ∈ D. inputs = inputs_0 + w }
      progress  { inputs with fun i i' => ∃w ∈ D. i' = i +' ('+w+')' }
      CONT();
      if(n < n0) then return else (aux(); if(pc = ∞) then return)
    done
  with Diverges (pc_i,n_i,inputs_i)_{i∈ℕ} -> ()
  end-try

let main()
  requires { pc = 0 }
  ensures  { pc' = 2 ∨ pc' = ∞ }
  ensures  { pc' = 2 → ∃w ∈ D. inputs' = inputs + w+')' }
  ensures  { pc' = ∞ →
    (∃(w_j)_{j∈ℕ}. inputs' = inputs + Σ_{0≤j} (w_j +' ('))
    ∨ (∃N, (p_i)_{0≤i<N}, (w_j)_{j∈ℕ}.
        inputs' = inputs + Σ_{0≤i<N} (p_i +' (') + Σ_{0≤j} ('('+w_j+')'))
  }
= try CONT(); aux (); CONT()
  with Diverges (pc_i,n_i,inputs_i)_{i∈ℕ} -> ()
  end-try
```

Figure 2: Ghost debugger for verification of all behaviors of Dyck language parser

same reasoning principle if the progression relation is the inverse of a well-founded relation, as in that case such sequences cannot exist.

We give a ghost debugger following that methodology in Figure 2, using as progression relations the existence of a single block shaped as $'('+w+')'$ for loops and as $w +' ('$ for recursion. We verified this program using the Why3 tool. In order to encode the divergence handling mechanism, we added before the loop/call to recursively defined function a non-deterministic code fragment that chooses in a non-deterministic manner either to do nothing or to simulate non-termination of the subsequent program construction, by assuming an infinite sequence of states with the property mentioned above.

Finally, note that for simplicity the code given in Figure 2 omits a few administrative details. First, we need non-trivial handlers for divergence in order to carry out an effective proof in Why3, which supplies proof indications to the automated theorem provers (but does not affect the state). Second, we need to ensure that non-terminating executions of the ghost debugger indeed map to non-terminating execution of the original program. In itself, mapping non-terminating execution to terminating execution is not a problem for the verification procedure, as the state of all variables is then well-defined upon divergence, and the ghost debugger may even perform subsequent calls to `CONT()`. However, we want divergent behaviors to match here, as we need to obtain a divergent status ($pc' = \infty$) in the post-condition. We achieve that goal by adding a timer to the status variables, which increase by 1 each time `CONT()` is

called. We then characterize effective divergence by testing whether the timer is unbounded over the state sequence, which is obviously the case here.

2.3 Example: Proving Simulation between Source and Assembly

Our last example shows how our approach applies to verifying simulation between two programs, more precisely the input and output of a compilation pass. The core idea is that our ghost debugger program can be used to verify multiple programs at once, in which case the transfer property will give us a simulation result. For verifying the result of a compilation, this amounts to have state variables for both programs, and two procedures to update the state of respective programs. Indeed, the ghost debugger is then equivalent to a cascade of interleaved calls to the two respective update functions. As those calls are independent, this is equivalent to performing them in parallel. We then claim the following transfer property: any property proved in a contract of the ghost debugger is also true for simultaneous executions of the source and compiled programs. In particular, if the ghost debugger prove that executing both in parallel with equal inputs eventually leads to a terminating state with equal output, or non-terminating state with equal traces of observable effects, then the two programs must behave the same on equal inputs. This means that such a ghost debugger can be used as a compilation certificate.

We focus here on a simple case, the compilation of lazy Boolean conjunction b_1 && b_2 to a stack machine. We assume that the stack machine state is made of a code pointer, a stack, and a store assigning values to variable. We consider here a compilation schema that map Boolean expressions to sequence of instructions that jump to label $ExitTrue$ if the expression is true, and to label $ExitFalse$ otherwise. In that setting, we can compile Boolean conjunction b_1 && b_2 directly as:

```
S: c1
M: c2
```

where c_1 jump to M or $ExitFalse$ when b_1 evaluates to true or false, respectively, and c_2 jumps to $ExitTrue$ or $ExitFalse$ when b_2 evaluates to true or false, respectively.

We represent the state of the compiled program by three variables pc, $stack$, and $store$, corresponding to the code pointer, the stack and the store of the machine. For the source program, we represent the state as an environment env assigning values to variables, and a decomposition of the code to be executed as an evaluation context $evalContext$ and an evaluation focus $evalFocus$. The evaluation context represent what should be executed once $evalFocus$ has finished executing. The evaluation focus can either be an effective fragment of the source program, in which case the next step is to analyze that fragment to execute it, or the result of an intermediate operation, in which case the next step is to recover the next step from the context.

To update the state, we introduce two abstract procedures `STEP_SRC` and `STEP_TGT` which respectively mimic the small-step semantics of the source and of the target language. We do not provide their specification as they are both extremely verbose and not informative. We can then prove compilation of lazy conjunction b_1 && b_2 the following way. First, we introduce abstract procedures `proof_1` and `proof_2` that represent the simultaneous execution of b_1/c_1 and b_2/c_2 correctly with respect to the chosen compilation schema. In other words, we suppose that `proof_1` and `proof_2` are already existing compilation certificates for sub-expressions. Then, we build a ghost debugger `proof_conj` that represent the simultaneous execution of b_1 && b_2 and its compiled pendant, and prove that the simultaneous execution respect the compilation schema. The full ghost debugger is given in Figure 3. This is verified nearly as-is (modulo syntactic details) by Why3. Note that we do not need to use `STEP_TGT` as the code to process conjunction is entirely included in the compiled program for sub-expression, while we need `STEP_SRC` to perform administrative context changes in the source program. However, we would use `STEP_TGT` to prove cases where the compiled program includes extra instruction, like for addition.

Note that we consider that `proof_i` procedures are abstract here only because we do not look at the

```
val proof_1() : bool
  requires { store = env ∧ evalFocus = toEval(b1) ∧ pc = S }
  ensures  { pc' = if result then M else ExitFalse }
  ensures  { evalFocus' = evaluated(result) ∧ evalContext' = evalContext }
  ensures  { stack' = stack ∧ store' = store ∧ env' = env }

val proof_2() : bool
  requires { store = env ∧ evalFocus = toEval(b2) ∧ pc = M }
  ensures  { pc' = if result then ExitTrue else ExitFalse }
  ensures  { evalFocus' = evaluated(result) ∧ evalContext' = evalContext }
  ensures  { stack' = stack ∧ store' = store ∧ env' = env }

let proof_conj() : bool
  requires { store = env ∧ evalFocus = toEval(b1 && b2) }
  ensures  { pc' = if result then ExitTrue else ExitFalse }
  ensures  { evalFocus' = evaluated(result) ∧ evalContext' = evalContext }
  ensures  { stack' = stack ∧ store' = store ∧ env' = env }
= STEP_SRC();
  let b = proof_1() in
  STEP_SRC(); b && proof_2()
```

Figure 3: Ghost debugger for simulation between source/compiled program (lazy conjunction case)

content of the b_i/c_i. In order to prove a complete simulation result, we need to interface them to actual ghost debugger, which depends on the actual b_i/c_i. In particular, notice that we can carry out the proof by proving components for each compilation case, and then interface them together. We claim that using this technique, we can not only provide certificates for the full compilation but also verify statically the compiler. The method is to prove the components statically, then use them to prove that the compiler necessarily create a ghost debugger that satisfy the intended contract. In particular, the compiler do not need to create that ghost debugger, so it can be represented as ghost data. This method is closely related to the proof method followed for the proof of the CakeML compiler [5], which exploit a program logic for the machine code.

If we wanted to use such a methodology to prove the correctness of an effective compiler, we would also need to prove preservation of divergent behaviors. We can achieve that objective by exploiting the divergence handling mechanism introduced in Section 2.2, which let us match divergent execution traces for the source and compiled programs. We expect that this methodology should be especially interesting to certify compilation of (mutually) recursive functions, for which we can use a ghost debugger with recursive structure. We expect to be able to exploit the same stack elimination phenomenon as for the previous examples, hence avoiding to state relational properties between execution stacks.

Finally, we remark that with a few changes, the methodology presented here can support non-determinism in the source or the compiled program. Indeed, the desired simulation property in presence of non-determinism is that any behavior of the compiled program is a behavior of the source program. In other words, we need to be able to re-construct an execution of the source program from an execution of the compiled program. We can achieve a transfer property that corresponds exactly to such execution reconstruction by adding arguments to the `STEP_SRC` procedure, which determines the non-deterministic decision for the source program. This mean that the non-deterministic decisions for the source language are now taken from the ghost debugger, and not by the execution of the source program. In particular, this leads to a transfer property of existential nature with respect to the source program.

If we view again the ghost debugger as an interleaving of calls to `STEP_SRC` and `STEP_TGT`, we cannot

fully re-order the calls in that setting as `STEP_SRC` has arguments which typically depends on the behavior of previous calls to `STEP_TGT`. However, we can re-order the calls so that all calls to `STEP_TGT` happens before the calls to `STEP_SRC`. Due to our divergence handling mechanisms, this makes sense even if the number of calls to `STEP_TGT` is infinite. Then, the ghost debugger become equivalent to executing the compiled program up to some point (or non-termination), then build a selected execution trace for the source program, in a manner that depends of the compiled execution. In particular, we can transfer contracts proved for the ghost debugger to trace transfer properties.

3 A Game-Theoretic Program Logic

3.1 Games

We base our verification approach on the formal model of *games*, which can be thought as the natural generalization of transition systems in the presence of both angelic and demonic non-determinism. In this setting, the validity of a Hoare triple corresponds to the existence of a winning strategy for the angel. As we solely focus on tools to prove the existence of such winning strategies for the angel, we define games asymmetrically with respect to both players. The domain of the game corresponds to angelic states, while demonic states are implicitly defined to be sets of of angelic states. We made those choices to match the behavior of the 'continue' routine presented earlier. Angelic choice selects the arguments given to that routine, while demonic choice selects the behavior of that routine among those allowed by the arguments.

In order to handle the case of non-terminating programs with observable effects, we equip our games with an order, so that the state only grows during a run of the game. We then represent the overall infinite execution of a program by the least upper bound of a run, which is typically the sequence of all observable effects. We impose that the order is chain-complete so that the requested least upper bound always exists.

Definition 3.1 (Chain-complete order) *A partially ordered set $(O, \leq)$ is* chain-complete *if for all non-empty subset X of O totally ordered by $\leq$, X admits a least upper bound.*

Definition 3.2 (Game) *A* game *$\mathbb{G} = (G, \preccurlyeq, \Delta)$ is a chain-complete partially ordered set $(G, \preccurlyeq)$ equipped with a function Δ from G to $\mathcal{P}(\mathcal{P}(G))$ such that*

$$\forall x \in G, \forall X \in \Delta(x), \forall y \in X, x \prec y$$

We respectively denote G, $\preccurlyeq$ and Δ as the domain*, the* order *and the* transitions *of the game.*

Definition 3.3 (History) *Given a game $\mathbb{G} = (G, \preccurlyeq, \Delta)$, an* history *of $\mathbb{G}$ is a non-empty subset of G for which the restriction of $\preccurlyeq$ is a total order, and which admits a maximum for that order.*

Definition 3.4 (Prefix Order) *Given two histories H_1, H_2 for a game $\mathbb{G} = (G, \preccurlyeq, \Delta)$, we say that H_1 is a prefix of H_2 if $H_1 \subseteq H_2$ and for all $x \in H_2 \backslash H_1$, x is an upper bound of H_1.*

Definition 3.5 (Victory Invariant) *Given a game $\mathbb{G} = (G, \preccurlyeq, \Delta)$ and two subsets P, Q of G, a* victory invariant *of $\mathbb{G}$ for P, Q is a set I of histories of $\mathbb{G}$ such that*

(i) for all $x \in P$, $\{x\} \in I$

(ii) for all $H \in I$ such that $H \cap Q = \emptyset$, there exists $X \in \Delta(\max H)$ such that for all $x \in X$, $H \cup \{x\} \in I$

(iii) for all non-empty $\mathcal{H} \subseteq I$ which is totally ordered by the prefix order, $\bigcup \mathcal{H} \cup \{\sup \mathcal{H}\} \in I$

The notion of victory invariant and history corresponds naturally to winning strategy with memories, where the initial state of the game is chosen by the demonic player in P, and the winning states are given by Q. Indeed, an history is exactly the memory of a play, minus the angelic moves that the angel can reconstruct. Moreover, the three propagation rules of victory invariants tells us exactly how to build a winning strategy. Rule (i) means that such a strategy can be built when starting from any element of P. Rule (ii) means that when the current play is H, either the angel has already won the game or a (not necessarily unique) winning decision can be made at that point. Finally, rule (iii) means that if a play grows indefinitely, it continues at the least upper bound. Note that building a play while respecting the invariant guarantees that after some ordinal number of iterations, the winning set Q will be attained. This is a consequence of the fact that the play is extended by a new element at each iteration, and that an ordinal with cardinal strictly bigger than G will eventually be reached.

We prefer using the notion of victory invariant rather than the equivalent notion of strategy for two reasons. First, victory invariants are a bit more permissive as they can allow several distinct angelic moves from a single history. In particular, they are more practical to perform proofs. Second, victory invariants are quite similar to loop invariants, which makes them closely related to program logics. In fact, the existence of a loop invariant for P, Q is the natural generalization of the validity of a Hoare triple, with the code being the game.

Remark: One might wonder why we did not define a notion of memoryless winning strategy instead, as move assignment functions from game states. Indeed, in a lot of cases we can prove that memoryless strategies have the same power as their memory-using variants. For example, this is the case when the game has no choice for one of the player, or when the winning set Q contains either all the states reachable after an infinite number of iteration, or when Q contains none of them. Note that the last two cases map naturally to partial and total correctness of programs. However, we have not been able to determine whether in general, we could derive a memoryless strategy from a memory-using one, as the absence of memory could prevent the strategy from pushing the state toward the correct least upper bound. Since our logic can lead to the construction of all victory invariants, hence all strategies, we use memory.

3.2 A Minimalist Verification Language

We now define the language $\mathcal{W}_G$ that we use to establish victory invariant on games $\mathbb{G} = (G, \preccurlyeq, \Delta)$. This is a mostly functional programming language, where functions are first-order and equipped with contracts. The only non-functional feature is a global variable $\texttt{now} \in G$, which represents the current state of the game. We only permit update of this variable through the use of predefined functions, so that the evolution of now respects the structure of the game. The prototypical predefined update function is $\texttt{step}^{\{0\}}_{\mathcal{P}(G)}$, which perform one step of a run. This procedure take as argument an element X of $\Delta(\texttt{now})$ and update the global variable to a non-deterministic element of X.

The syntax of the language $\mathcal{W}_G$ is given in Figure 4. In these definitions, $expr$ and $formula$ refer respectively to arbitrary mathematical expressions and formulas in some meta-logic language, whose exact nature is irrelevant here. Those expressions can access any immutable variable, as well as the global variable now. Formulas in contract post-conditions can also access the special variable old, which refers to the state of the global variable now before the execution of a function.

The syntactic categories var and fun corresponds respectively to immutable variable and function names. In order to make the typing rules as simple as possible, we tag each immutable variable x_A with its type A, which we defines to be a non-empty set. Similarly, function names $\texttt{f}^B_A$ are tagged with their input type A and output type B. As the language contains continuations procedures, the output type of a function is allowed to be empty. We define the typing system for language $\mathcal{W}_G$ as a predicate $\vdash \pi : B$, which means that π is well-typed with type B. The typing rules are given in Figure 5. Those rules depend on the typing relations $\vdash e : A$ and $\vdash \varphi$ for expressions and formulas of the meta-logic. We use notation $\vdash_{\texttt{old}} \varphi$ instead to indicate when old is allowed in formulas. Note that by straightforward induction, for

$$
\begin{array}{lll}
pgm & ::= & fun(expr) \\
 & | & \mathsf{let}\ var = pgm\ \mathsf{in}\ pgm \\
 & | & \mathsf{switch}\ case^*\ \mathsf{end\text{-}switch} \\
 & | & \mathsf{kont}\ fun\ \mathsf{in}\ pgm\ \mathsf{end\text{-}kont} \\
 & | & \mathsf{rec}\ fun(var) : contract \\
 & & \quad \mathsf{progress}(expr, expr) \\
 & & \quad = pgm \\
 & & \quad \mathsf{catch}(var, var, var) : \\
 & & \quad\quad pgm \\
 & & \mathsf{in}\ pgm \\
case & ::= & \mathsf{case}\ var : formula\ \mathsf{in}\ pgm \\
contract & ::= & \langle formula \hookrightarrow var.\ formula \rangle
\end{array}
$$

Figure 4: Definition of $\mathcal{W}_G$

each well-typed program π there exists a minimal type B for which $\vdash \pi : B$, in the sense that the other typing judgments can be derived by the casting rule for programs with empty output type. We call that type the output type of π.

We only give one unusual construction for our verification language: the recursive construction, which introduce a recursively-defined function equipped with a handler to process the result of diverging behavior. The progress clause gives a pair r, e of a fixed order r and of an expression e in the function parameters, such that e must increase with respect to r at each recursive call. This recursive construction behave as usual recursive function during finite executions, but has different behavior for infinite one. If the recursion stack grows infinitely, the divergence handling code is called with three arguments representing precisely the recursion stack:

- First, the set of values H taken by e along the diverging recursion stack. Since e can only increase, this set gives the structure of the diverging recursion stack.
- Second, a mapping from H to the function parameters along the diverging recursion stack
- Third, a mapping from H to the values taken by the global variable `now` along the diverging recursion stack

The task of the divergence handler is to send back one of the calls in the recursion stack along the intended return value for that call. Once it does that, execution of the recursive function resume as if the corresponding call had terminated with that return value. Note that the divergence handler is allowed to make recursive calls for that purpose as long as e increase even further, which may lead to divergence handler being called recursively with even larger recursion stacks. In particular, note that due to such nested divergence the set H representing the stack may follow the structure of arbitrary limit ordinals, including uncountable ones.

Note that we choose to keep our verification language as small as possible, in order to keep proofs simple. We notably do not provide loops, conditionals, assertions, and bare mathematical expressions, as we can derive them from the core constructions of the language. For example, we can encode a program $(e)_B$ returning the value of the expression e with type B as $(\mathsf{kont}\ k_B^\emptyset\ \mathsf{in}\ k_B^\emptyset(e)\ \mathsf{end\text{-}kont})$.

Remark: We give a definition of recursive functions that seems very different than the one we used in Section 2.2. In the example, the divergence handler was used outside the recursive definition, and the progression relation was not restricted to an order. We claim that this construction is derivable. Indeed, we recover an external divergence handling mechanism by taking a continuation before-hand to jump out of

$$\frac{\vdash e : A}{\vdash \mathsf{f}_A^B(e) : B} \qquad \frac{\vdash \pi_0 : A \qquad \vdash \pi_1 : B}{\vdash \mathsf{let}\ x_A = \pi_0\ \mathsf{in}\ \pi_1 : B} \qquad \frac{\vdash \pi : \emptyset}{\vdash \pi : B}$$

$$\frac{\vdash \pi : B \qquad \vdash \varphi}{\vdash \mathsf{case}\ x_A : \varphi\ \mathsf{in}\ \pi : B} \qquad \frac{\vdash \pi : B}{\vdash \mathsf{kont}\ \mathsf{f}_A^\emptyset\ \mathsf{in}\ \pi\ \mathsf{end\text{-}kont} : B}$$

$$\frac{\forall i \in [\![1; n]\!] \vdash c_i : B}{\vdash \mathsf{switch}\ c_1 \dots c_n\ \mathsf{end\text{-}switch} : B}$$

$$\frac{\vdash \pi_0 : B \quad \vdash \pi_1 : U \times B \quad \vdash \pi_2 : C \quad \vdash r : \mathcal{P}(U \times U) \quad \vdash e : U \quad \vdash \varphi_0 \quad \vdash_{\mathtt{old}} \varphi_1}{\vdash \begin{pmatrix} \mathsf{rec}\ \mathsf{f}_A^B(x_A) : \langle \varphi_0 \hookrightarrow y_B.\ \varphi_1 \rangle \\ \quad \mathsf{progress}(r, e) \\ \quad = \pi_0 \\ \quad \mathsf{catch}(H_{\mathcal{P}(U)}, g_{A^U}, h_{G^U}) : \\ \qquad \pi_1 \\ \mathsf{in}\ \pi_2 \end{pmatrix} : C}$$

Figure 5: Typing rules for $\mathcal{W}_G$

the recursion from the handler, and using extra variables to remember the minimum level of the stack. We also need a minor addition to the pre-condition of the recursive definition to remember that the minimum level is indeed minimal, so we can recover that information in the handler. We then recover a construction with the same behavior as the one used by the example, except for the restriction of progression relation to orders. Similarly, we can lift that last restriction via another derived construction, which replaces the parameter of the recursive function by the finite sequence of its values till the bottom of the stack, and the progression relation by sequence extension. Then, we can recover easily the step-by-step progression relation by requiring it for the sequence argument of the recursive function. Note that this second step prevents recursive calls in the divergence handler, which is typically not a problem.

3.3 Predicate Transformer Semantics for $\mathcal{W}_G$

We now give predicate transformer semantics for $\mathcal{W}_G$, using weakest pre-condition transformers in the style of Dijkstra [3]. Intuitively, the transformer of a program π for a winning set Q gives the largest set P such that π proves the existence of a victory invariant for P, Q.

Due to the presence of continuations as well as defined objects, we parameterize weakest pre-condition transformer by contextual elements. We add an assignation parameter, which gives the values of variables, and a context parameter, which gives the specification of functions.

Definition 3.6 (Assignation) *An* assignation *for $\mathcal{W}_G$ is a mapping σ from variables names x_A to values in the set A tagging the variable name.*

Definition 3.7 (Context) *A* context *for $\mathcal{W}_G$ is a mapping Γ from function names f_A^B to pairs $P \subseteq A \times G, Q \subseteq A \times G \times B \times G$. Intuitively, the pair $P, Q = \Gamma(\mathsf{f}_A^B)$ represents the* contract *for procedure f_A^B within context Γ.*

Remark: We choose to make contexts and assignations total in order to eliminate spurious interactions between the scope of variables and the semantics of $\mathcal{W}_G$. This is also the reason for which we impose variables to be tagged by non-empty sets, as otherwise an assignation might not exists. However,

there is a subtlety as the domains of those contextual objects are too big to be sets. For this reason we implicitly suppose that the sets that can be used as types are restricted to the members of a "universe" set. This restriction has no impact in practice since we can choose the universe as the set of types that occurs in a given program.

We then define the semantics of $\mathcal{W}_G$ as a mapping $\mathcal{T}[\Gamma, \sigma]\,(\pi, Q)$, which takes as argument a context Γ, an assignation σ, a well-typed program π with output type B, and a set $Q \subseteq B \times G$ (the post-condition), and sends back a subset of G (the weakest pre-condition). The expressions and formulas within the program are interpreted by the mean of a standard interpretation function $[\![\cdots]\!]$..., which given an expression/a formula and an assignation of its variables compute the value of the expression as an element of its type, or the satisfaction of the formula as a truth value. In order to handle the casts of $\emptyset$ to B, we make a small distinction for program with output type B, via definition $\mathcal{T}'[\Gamma, \sigma]\,(\pi, Q)$. We also define a predicate $\mathcal{C}[(\Gamma_x)_{x\in G}, \sigma]\,(\pi : \langle \varphi_0 \hookrightarrow y_B.\ \varphi_1\rangle)$ corresponding to the notion of contract satisfaction, which we use for recursion. We give the full definition in Figure 6.

The definition of the backward predicate transformer semantics for $\mathcal{W}_G$ is mostly the standard definition of weakest pre-conditions, except for two cases: continuation and recursion. We define the transformer for continuation introduction by adapting naturally the typing schema of the `call-with-current-continuation` operator. For introduction of recursive function, we perform the following modifications to the traditional transformer for total correctness:

- We replace the traditional well-founded decreasing termination measure by a progression order, which must increase along recursive call. We do not require any analog of well-foundedness as allowing non-termination is desirable in our case.

- We add condition (iv) for the divergence handling code π_1. This condition states that π_1 terminates correctly given any limit recursion stack controlled by the pre-condition and the progression relation of the program. Note that we do not impose a well-foundedness criterion on the stack, as we expect it to be an unnecessary constraint in most case. Moreover, this extra constraint can be enforced through a derived construction.

3.4 Transfer Property: from Weakest Pre-Condition to Victory Invariants

We now state the main correctness result for our verification language, as a transfer property from language $\mathcal{W}_G$ to games. Essentially, we claim that weakest pre-conditions for $\mathcal{W}_G$ induce the existence of victory invariants.

Definition 3.8 *We define the* step contract $(P_{\mathbb{G}}, Q_{\mathbb{G}})$ *associated to game* $\mathbb{G} = (G, \preccurlyeq, \Delta)$ *as*

$$P_{\mathbb{G}} = \{(X, x) \in \mathcal{P}(G) \times G \mid X \in \Delta(x)\}$$
$$Q_{\mathbb{G}} = \{(X, x, 0, y) \in \mathcal{P}(G) \times G \times \{0\} \times G \mid y \in X\}$$

Definition 3.9 *We define the* step context $\Gamma_{\mathbb{G}}$ *associated to game* $\mathbb{G} = (G, \preccurlyeq, \Delta)$ *as the context mapping* $\mathtt{step}^{\{0\}}_{\mathcal{P}(G)}$ *to the step contract* $(P_{\mathbb{G}}, Q_{\mathbb{G}})$ *of* $\mathbb{G}$*, and other function names to the empty contract* $(\emptyset, \emptyset)$*.*

Theorem 3.10 *For all games* $\mathbb{G} = (G, \preccurlyeq, \Delta)$*, all well-typed programs* π *of* $\mathcal{W}_G$ *with output type* $\{0\}$*, all subsets* Q *of* G*, and all assignations* σ*, there is a victory invariant for* $\mathcal{T}[\Gamma_{\mathbb{G}}, \sigma]\,(\pi, \{0\} \times Q)\,, Q$*.*

We derive from theorem 3.10 an immediate corollary, which conclude the existence of a victory invariant for a pair of sets definable in the meta-logic as long as we can find a program satisfying the corresponding contract.

$$\mathcal{T}'[\Gamma,\sigma]\,(\pi,Q) \triangleq \begin{cases} \mathcal{T}[\Gamma,\sigma]\,(\pi,\emptyset) & \text{if output type of } \pi \text{ is } \emptyset \\ \mathcal{T}[\Gamma,\sigma]\,(\pi,Q) & \text{otherwise} \end{cases}$$

$$\mathcal{T}[\Gamma,\sigma]\left(\mathsf{f}_A^B(e),Q\right) \triangleq \{x \in G \mid (a_x,x) \in P_f \wedge \forall b,y.\ (a_x,x,b,y) \in Q_f \Rightarrow (b,y) \in Q\}$$
$$\text{where } (P_f,Q_f) = \Gamma(\mathsf{f}_A^B) \text{ and } a_x = [\![e]\!]_{\sigma,\mathsf{now}\leftarrow x}$$

$$\mathcal{T}[\Gamma,\sigma]\,(\mathsf{let}\ x_A = \pi_0\ \mathsf{in}\ \pi_1,Q) \triangleq \mathcal{T}'[\Gamma,\sigma]\,(\pi_0,\{(a,y) \in A \times G \mid y \in \mathcal{T}'[\Gamma,\sigma[x_A \leftarrow a]]\,(\pi_1,Q)\})$$

$$\mathcal{T}[\Gamma,\sigma]\left(\mathsf{kont}\ \mathsf{f}_A^\emptyset\ \mathsf{in}\ \pi\ \mathsf{end\text{-}kont},Q\right) \triangleq \mathcal{T}'[\Gamma[\mathsf{f}_A^\emptyset \leftarrow (Q,\emptyset)],\sigma]\,(\pi,Q)$$

$$\mathcal{T}[\Gamma,\sigma]\left(\mathsf{case}\ x_A : \varphi\ \mathsf{in}\ \pi,Q\right) \triangleq \bigcap_{a\in A} \{x \in G \mid [\![\varphi]\!]_{\sigma[x_A\leftarrow a],\mathsf{now}\leftarrow x} \Rightarrow x \in \mathcal{T}'[\Gamma,\sigma[x_A \leftarrow a]]\,(\pi,Q)\}$$

$$Grd[\sigma](\mathsf{case}\ x_A : \varphi\ \mathsf{in}\ \pi) \triangleq \bigcup_{a\in A} \{x \in G \mid [\![\varphi]\!]_{\sigma[x_A\leftarrow a],\mathsf{now}\leftarrow x}\}$$

$$\mathcal{T}[\Gamma,\sigma]\,(\mathsf{switch}\ c_1 \dots c_n\ \mathsf{end\text{-}switch},Q) \triangleq \left(\bigcup_{i\in[\![1;n]\!]} Grd[\sigma](c_i)\right) \cap \bigcap_{i\in[\![1;n]\!]} \mathcal{T}[\Gamma,\sigma]\,(c_i,Q)$$

$$\mathcal{C}[(\Gamma_x)_{x\in G},\sigma]\,(\pi : \langle\varphi_0 \hookrightarrow y_B.\ \varphi_1\rangle) \triangleq \forall x \in G.\ [\![\varphi_0]\!]_{\sigma,\mathsf{now}\leftarrow x} \Rightarrow x \in \mathcal{T}'[\Gamma_x,\sigma]\,(\pi,Q_x)$$
$$\text{where } Q_x = \{(b,y) \in B \times G \mid [\![\varphi_1]\!]_{\sigma[y_B\leftarrow b],\mathsf{now}\leftarrow y,\mathtt{old}\leftarrow x}\}$$

$$z \in \mathcal{T}[\Gamma,\sigma]\left(\begin{array}{l} \mathsf{rec}\ \mathsf{f}_A^B(x_A) : \langle\varphi_0 \hookrightarrow y_B.\ \varphi_1\rangle \\ \quad \mathsf{progress}(r,e) \\ \quad = \pi_0 \\ \quad \mathsf{catch}(H_{\mathcal{P}(U)},h_{A^U},h_{G^U}) :' Q \\ \qquad \pi_1 \\ \mathsf{in}\ \pi_2 \end{array}\right) \text{ if and only if following conditions (i)-(iv) hold:}$$

(i) $[\![r]\!]_{\sigma,\mathsf{now}\leftarrow z}$ is a strict order
(ii) $\forall a \in A.\ \mathcal{C}[(\Gamma^{rec}_{\{[\![e]\!]_{\sigma[x_A\leftarrow a],\mathsf{now}\leftarrow x}\}})_{x\in G},\sigma[x_A \leftarrow a]]\,(\pi_0 : \langle\varphi_0 \hookrightarrow y_B.\ \varphi_1\rangle)$
(iii) $\mathcal{T}'[\Gamma^{rec}_\emptyset,\sigma]\,(\pi_2,Q)$
(iv) for all $H \in \mathcal{P}(U), m_A \in A, m_G \in G^U$ such that:
  (a) H is totally ordered by relation $[\![r]\!]_{\sigma,\mathsf{now}\leftarrow z}$, and does not have a maximum
  (b) H is inhabited
  (c) m_G is an increasing function from $(H,[\![r]\!]_{\sigma,\mathsf{now}\leftarrow z})$ to $(G,\preccurlyeq)$
  (d) $\forall u \in H.\ [\![\varphi_0]\!]_{\sigma[x_A\leftarrow m_A(u)],\mathsf{now}\leftarrow m_G(u)}$
  then $\sup_{u\in H} m_G(u) \in \mathcal{T}'[\Gamma^{rec}_H,\sigma[H_{\mathcal{P}(U)},h_{A^U},h_{G^U} \leftarrow H,m_A,m_G]]\left(\pi_1,Q^{lim}_{H,m_A,m_G}\right)$
where $\Gamma^{rec}_X = \Gamma[\mathsf{f}_A^B \leftarrow (P^{rec}_X,Q^{step})]$
and $P^{rec}_X = \{(a,x) \in A \times G \mid [\![\varphi_0]\!]_{\sigma[x_A\leftarrow A],\mathsf{now}\leftarrow x} \wedge \forall u \in X.\ u[\![r]\!]_{\sigma,\mathsf{now}\leftarrow z}[\![e]\!]_{\sigma[x_A\leftarrow a],\mathsf{now}\leftarrow x}\}$
and $Q^{step} = \{(a,x,b,y) \in A \times G \times B \times G \mid [\![\varphi_1]\!]_{\sigma[x_A\leftarrow a][y_B\leftarrow b],\mathsf{now}\leftarrow y,\mathtt{old}\leftarrow x}\}$
and $Q^{lim}_{H,m_A,m_G} = \{((u,b),y) \in (U \times B) \times G \mid u \in H \wedge [\![\varphi_1]\!]_{\sigma[x_A\leftarrow m_A(u)][y_B\leftarrow b],\mathsf{now}\leftarrow y,\mathtt{old}\leftarrow m_G(u)}\}$

Figure 6: Predicate transformer semantics for $\mathcal{W}_G$

Corollary 3.11 *For all games $\mathbb{G} = (G, \preccurlyeq, \Delta)$, all well-typed programs π of $\mathcal{W}_G$ with output type $\{0\}$, all well-typed formula pairs φ_0, φ_1 not mentioning* `old`*, and all assignations σ, if the contract satisfaction property $\mathcal{C}[(\Gamma_{\mathbb{G}})_{x\in G}, \sigma]\left(\pi : \left\langle \varphi_0 \hookrightarrow u_{\{0\}}.\ \varphi_1 \right\rangle\right)$ holds, then there is a victory invariant for the set pair $\{x \in G \mid [\![\varphi_0]\!]_{\sigma,\mathsf{now}\leftarrow x}\}, \{x \in G \mid [\![\varphi_1]\!]_{\sigma,\mathsf{now}\leftarrow x}\}$.*

In order to prove theorem 3.10, we first need to generalize it over both contexts and output types.

Definition 3.12 *A contract $P \subseteq A \times G, Q \subseteq A \times G \times B \times G$ is said to be* valid *with respect to game $\mathbb{G} = (G, \preccurlyeq, \Delta)$ if for all $(a, x) \in A \times G$, there exists a victory invariant for the set pair $\{x\}, \{y \in G \mid \exists b \in B.\ (a, x, b, y) \in Q\}$. By extension, a context Γ is* valid *with respect to $\mathbb{G}$ if all its contracts are valid with respect to $\mathbb{G}$.*

Theorem 3.13 *For all games $\mathbb{G}$, all well-typed programs π of $\mathcal{W}_G$ with output type B, all contexts Γ valid with respect to $\mathbb{G}$, all subsets Q of $B \times G$, and all assignations σ, there is a victory invariant for the pair $\mathcal{T}[\Gamma, \sigma]\,(\pi, Q)\,, \{y \in G \mid \exists b \in B.\ (b, y) \in Q\}$.*

Note that theorem 3.10 is an immediate consequence of theorem 3.13. **Proof:** It suffices to prove that $\Gamma_{\mathbb{G}}$ is valid with respect to $\mathbb{G}$. By definition, we can check that the empty contract $\emptyset, \emptyset$ is valid with respect to any game, so we reduce further the proof to validity of contract $P_{\mathbb{G}}, Q_{\mathbb{G}}$. Let (X, x) be an arbitrary pair in $P_{\mathbb{G}}$. We conclude by noticing that the set $I = \{\{x\}\} \cup \{\{x, y\} \mid y \in X\}$ is a valid victory invariant for the desired pair. □

3.5 Proof of Transfer Property

We will only give the sketch of the proof of theorem 3.10, as the complete proof it too long for inclusion here. Most of the proof concern our divergence-catching recursive construction. For full details, the reader may refer to our Why3 development[3], which mechanize the proof of the Hoare logic variant corresponding to the weakest pre-condition calculus for $\mathcal{W}_G$. The mechanized logic rules corresponds precisely to the victory invariant constructions we need for the proof.

The proof proceed by induction over the syntax of $\mathcal{W}_G$. First, we note that following the structure of the definition of $\mathcal{T}[\Gamma, \sigma]\,(\pi, Q)$, we can replace $\mathcal{T}[\Gamma, \sigma]\,(\pi, Q)$ by $\mathcal{T}'[\Gamma, \sigma]\,(\pi, Q)$ in the induction hypothesis, since a victory invariant for empty winning set is also a victory invariant for any winning set.

Lemma 3.14 *For all games $\mathbb{G} = (G, \preccurlyeq, \Delta)$, P, Q, P', Q' subsets of G such that $P' \subseteq P$ and $Q \subseteq Q'$, any victory invariant for P, Q is a victory invariant for P', Q'.*

Proof: From the definition of victory invariant (cf definition 3.5), replacing P by P' weakens condition (i), and replacing Q' by Q strengthens the hypothesis of condition (ii). Hence it weakens the constraints for a set to be a victory invariant relative to that particular set pair. □

Second, we use a lemma which will let us focus the proof of existence of victory invariant for singleton subsets of $\mathcal{T}[\Gamma, \sigma]\,(\pi, Q)$ instead of the complete set.

Lemma 3.15 *For all games $\mathbb{G} = (G, \preccurlyeq, \Delta)$, P, Q subset of G, if for all $x \in P$ there exists a victory invariant I_x for $\{x\}, Q$, then there exists a victory invariant for P, Q.*

Proof: We cannot take directly the union $I = \bigcup_{x\in P} I_x$ as a victory invariant of P, Q because of condition (iii) (cf definition 3.5). The problem is that the history within a chain of I might not correspond to a consistent choice of x. We will use the minimum of history to distinguish the correct one. To that hand, let us define $J_x = \{H \in I_x \mid x \text{ is minimum of } H\}$. It is immediate to check that J_x is still a victory invariant for $\{x\}, Q$, as the propagation rules (ii) and (iii) preserve the minimum. Then, $J = \bigcup_{x\in P} J_x$

[3] `http://toccata.lri.fr/gallery/hoare_logic_and_games.en.html`

$$
\begin{array}{l}
\mathsf{rec}\ \mathsf{f}_A^{\{0\}}(x_A) : \langle Inv \hookrightarrow u_{\{0\}}.\ \bot \rangle \\
\quad \mathsf{progress}(r, e) \\
\quad = \mathsf{let}\ x_A = \pi_0\ \mathsf{in}\ \mathsf{f}_A^{\{0\}}(x_A) \\
\quad \mathsf{catch}(H_{\mathcal{P}(U)}, g_{A^U}, g_{G^U}) : \\
\qquad \mathsf{let}\ x_A = \pi_1\ \mathsf{in}\ \mathsf{f}_A^{\{0\}} \\
\mathsf{in\ let}\ u_{\{0\}} = \mathsf{f}_A^{\{0\}}(x_A)\ \mathsf{in\ switch\ \ end\text{-}switch}
\end{array}
$$

Figure 7: Encoding of iteration

is a victory invariant for P, Q. Condition (i) and (ii) are checked by straightforward case analysis. For condition (iii), it suffice to remark that for all non-empty $\mathcal{H} \subseteq J$ totally ordered by the prefix order, the elements of $\mathcal{H}$ have the same minimum element. In particular, $\mathcal{H} \subseteq J_x$ for some $x \in P$, so condition (iii) holds as well. □

We then focus on the simplest cases of the induction: function application, case analysis, and continuation introduction. **Proof:**

- Function application $\pi = \mathsf{f}_A^B(e)$: using lemma 3.15 and 3.14, we reduce to a condition that match exactly the validity of the function contract, which is true by hypothesis.
- Case analysis $\pi = \mathsf{switch}\ c_1 \dots c_n\ \mathsf{end\text{-}switch}$: we first use lemma 3.15 to focus on a single $x \in \mathcal{T}[\Gamma, \sigma]\,(\pi, Q)$. By definition, there exists some $i \in [\![1; n]\!]$ such that $x \in Grd[\sigma](c_i)$. Suppose that $c_i = \mathsf{case}\ x_A : \varphi\ \mathsf{in}\ \pi_i$. By definition, there exists $a \in A$ such that $[\![\varphi]\!]_{\sigma[x_A \leftarrow a], \mathsf{now} \leftarrow x}$ holds. But since $x \in \mathcal{T}[\Gamma, \sigma]\,(c_i, Q)$, $x \in \mathcal{T}'[\Gamma, \sigma[x_A \leftarrow a]]\,(\pi_i, Q)$ as well. By induction hypothesis and monotonicity (lemma 3.14), we derive the victory invariant for the set pair $\mathcal{T}[\Gamma, \sigma]\,(\pi, Q)\,, \{y \in G \mid \exists b \in B.\ (b, y) \in Q\}$.
- Continuation introduction $\pi = \mathsf{kont}\ \mathsf{k}_A^{\emptyset}\ \mathsf{in}\ \pi_0\ \mathsf{end\text{-}kont}$: note that since induction proceed over the program π, we can change the structure of the game $\mathbb{G} = (G, \preccurlyeq, \Delta)$ so that the contract that we add for $\mathsf{k}_A^{\emptyset}$ holds. We consider the game $\mathbb{G}' = (G, \preccurlyeq, \Delta')$ with

  $$\Delta'(x) = \begin{cases} \Delta(x) \cup \{\emptyset\} & \text{if } x \in Q \\ \Delta(x) & \text{otherwise} \end{cases}$$

  Relatively to $\mathbb{G}'$, contract $(Q, \emptyset)$ is immediately validated by the victory invariant $\{\{y\} \mid \exists b \in B.(b, y) \in Q\}$. Moreover, since $\mathbb{G}'$ has more transitions than $\mathbb{G}$, any victory invariant relative to $\mathbb{G}$ is a victory invariant relative to $\mathbb{G}'$, so contracts from Γ are still valid relatively to $\mathbb{G}'$. By induction hypothesis, we obtain exactly the desired victory invariant for the set pair $\mathcal{T}[\Gamma, \sigma]\,(\pi, Q)\,, \{y \in G \mid \exists b \in B.\ (b, y) \in Q\}$, except that it is relative to $\mathbb{G}'$ instead of $\mathbb{G}$. Finally, note that this victory invariant is also a victory invariant with respect to $\mathbb{G}$. Indeed, we cannot rely on one of the extra transitions in condition (ii) of definition 3.5, as this would contradict the hypothesis that history does not intersect post-condition.

□

For the sequential composition case, the natural solution would be to perform some form of concatenation of the victory invariants. As this can get quite complex, we choose to instead derive that case from the more general case of iteration, which we also use for the recursion case. We define the iterative construction as calling a non-returning, tail-recursive function immediately after its definition. We give the exact format in Figure 7. Note that due to continuations, the non-returning restriction can be easily enforced.

$$
\begin{array}{l}
\mathsf{kont}\ k_B^{\emptyset}\ \mathsf{in}\\
\quad \mathsf{let}\ x_{(\{0\}+A+B)\times\mathbb{N}} = (inj_1(0),0)_{(\{0\}+A+B)\times\mathbb{N}}\ \mathsf{in}\\
\quad \mathsf{rec}\ f^{\{0\}}_{(\{0\}+A+B)\times\mathbb{N}}(x_{(\{0\}+A+B)\times\mathbb{N}}) : \langle Inv \hookrightarrow u_{\{0\}}.\ \bot\rangle\\
\qquad \mathsf{progress}(<, proj_2(x_{(\{0\}+A+B)\times\mathbb{N}}))\\
\qquad = \mathsf{let}\ x_{(\{0\}+A+B)\times\mathbb{N}} =\\
\qquad\quad \mathsf{switch}\\
\qquad\quad \mathsf{case}\ u_{\{0\}} : inj_1(u_{\{0\}}) = proj_1(x_{(\{0\}+A+B)\times\mathbb{N}})\ \mathsf{in\ let}\ x_A = \pi_0\ \mathsf{in}\ (inj_2(x_A),1)_{(\{0\}+A+B)\times\mathbb{N}}\\
\qquad\quad \mathsf{case}\ x_A : inj_2(x_A) = proj_1(x_{(\{0\}+A+B)\times\mathbb{N}})\ \mathsf{in\ let}\ x_B = \pi_1\ \mathsf{in}\ (inj_3(x_B),2)_{(\{0\}+A+B)\times\mathbb{N}}\\
\qquad\quad \mathsf{case}\ x_B : inj_3(x_B) = proj_1(x_{(\{0\}+A+B)\times\mathbb{N}})\ \mathsf{in}\ k_B^{\emptyset}(x_B)\\
\qquad\quad \mathsf{end\text{-}switch}\\
\qquad \mathsf{in}\\
\qquad f^{\{0\}}_{(\{0\}+A+B)\times\mathbb{N}}(x_{(\{0\}+A+B)\times\mathbb{N}})\\
\qquad \mathsf{catch}(H_{\mathcal{P}(\mathbb{N})}, g_{((\{0\}+A+B)\times\mathbb{N})^{\mathbb{N}}}, g_{G^{\mathbb{N}}}) :\\
\qquad\quad \mathsf{let}\ x_{(\{0\}+A+B)\times\mathbb{N}} = \mathsf{switch\ \ end\text{-}switch\ in}\ f^{\{0\}}_{(\{0\}+A+B)\times\mathbb{N}}\\
\quad \mathsf{in\ let}\ u_{\{0\}} = f^{\{0\}}_{(\{0\}+A+B)\times\mathbb{N}}(x_{(\{0\}+A+B)\times\mathbb{N}})\ \mathsf{in\ switch\ \ end\text{-}switch}\\
\mathsf{end\text{-}kont}
\end{array}
$$

where $Inv = \begin{cases} \mathsf{now} \in \mathcal{T}[\Gamma,\sigma]\,(\pi,Q) & \text{if } x_{(\{0\}+A+B)\times\mathbb{N}} = (inj_1(0),0)\\ \mathsf{now} \in \mathcal{T}[\Gamma,\sigma[x_A \leftarrow a]]\,(\pi_1,Q) & \text{if } x_{(\{0\}+A+B)\times\mathbb{N}} = (inj_2(a),1)\\ (b,\mathsf{now}) \in Q & \text{if } x_{(\{0\}+A+B)\times\mathbb{N}} = (inj_3(b),2)\\ \bot & \text{otherwise} \end{cases}$

and $f^{\{0\}}_{(\{0\}+A+B)\times\mathbb{N}}, x_{(\{0\}+A+B)\times\mathbb{N}}, u_{\{0\}}$ do not occur in π

Figure 8: Reduction of general sequential composition $\pi = \mathsf{let}\ x_A = \pi_0\ \mathsf{in}\ \pi_1$ to iteration

We now provide the proof for sequential composition provided the iterative construction is correct, in the sense that whenever the two sub-programs π_0 and π_1 of an iterative construction instance satisfy the induction hypothesis, then the instance satisfy the same victory invariant existence property.

Proof: First, we prove that sequential composition $\pi = \mathsf{let}\ x_A = \pi_0\ \mathsf{in}\ \pi_1$ is correct when one of the two programs π_0 or π_1 is the encoding $(e)_C = \mathsf{kont}\ k_C^{\emptyset}\ \mathsf{in}\ k_C^{\emptyset}(e)\ \mathsf{end\text{-}kont}$ of a program defined by a pure expression e with output type C. We use lemma 3.15 to first focus on a single element $x \in \mathcal{T}[\Gamma,\sigma]\,(\pi,Q)$. In the case where the expression is π_0, the membership condition is actually equivalent to $x \in \mathcal{T}[\Gamma,\sigma[x_A \leftarrow [\![e]\!]_{\sigma,\mathsf{now}\leftarrow x}]\,(\pi_1,Q)$, that is, the expected update of the assignation. In particular, the induction hypothesis readily give us the desired victory invariant. In the other case, when the expression is π_1, the membership condition is equivalent to $x \in \mathcal{T}[\Gamma,\sigma]\left(\pi_0, \{(a,y) \in A\times G \mid [\![e]\!]_{\sigma[x_A\leftarrow a],\mathsf{now}\leftarrow y} \in Q\}\right)$, for which we can conclude by induction hypothesis as well.

Using all the constructions that we proved correct until now, included this restricted form sequential composition, as well as an instance of the iterative construction, we can now encode general sequential composition. The basic idea is that a sequence can be encoded by taking a continuation, then making a three-step iteration which calls the first program at first step, calls the second at second step, and the continuation at the third. More precisely, given the three parameters Γ,σ,Q of weakest pre-condition for $\pi = \mathsf{let}\ x_A = \pi_0\ \mathsf{in}\ \pi_1$, we can build a program using only π_0, π_1, and the construction already known/assumed correct, which has the same weakest pre-condition as π for the fixed parameters. In particular, we obtain immediately the desired victory invariant. We give the encoding in the case where $B \neq \emptyset$ in Figure 8, in which the operators inj_i and $proj_i$ are the natural injection/projection operators for

sum/products. The encoding for $B = \emptyset$ is essentially the same but without result wrapping around π_1, as we cannot declare x_B (and do not need to). □

It remains to deal with the recursive and iterative constructions, starting by the iterative one. We obtain it as consequence of a simulation result between games, which lift local transfer of transition and limit situations as victory invariant to global transfer of victory invariants.

Definition 3.16 *Given two games $\mathbb{G}_1 = (G_1, \preccurlyeq_1, \Delta_1)$ and $\mathbb{G}_2 = (G_2, \preccurlyeq_2, \Delta_2)$, we say that a relation $\mathcal{R} \subseteq G_1 \times G_2$ induce a simulation of $\mathbb{G}_1$ by $\mathbb{G}_2$ if for every pair P, Q for which a victory invariant exists in $\mathbb{G}_1$, a victory invariant exists for $\mathcal{R}[P], \mathcal{R}[Q]$ in $\mathbb{G}_2$.*

Definition 3.17 *Given two games $\mathbb{G}_1 = (G_1, \preccurlyeq_1, \Delta_1)$ and $\mathbb{G}_2 = (G_2, \preccurlyeq_2, \Delta_2)$, we say that a relation $\mathcal{R} \subseteq G_1 \times G_2$ induce a step-wise simulation of $\mathbb{G}_1$ by $\mathbb{G}_2$ if the following two condition holds:*

(i) For all $x \in G_1$, $X \in \Delta_1(x)$, there exists a victory invariant for $\mathcal{R}[\{x\}], \mathcal{R}[X]$ in $\mathbb{G}_2$.

(ii) For all H non-empty totally ordered subsets of G_1, for all monotonic functions f from $(H, \preccurlyeq_1)$ to $(G_2, \preccurlyeq_2)$ respecting $\mathcal{R}$, that is, $x\mathcal{R}f(x)$ for all $x \in H$, there is a victory invariant for pair $\{\sup_{x\in H} f(x)\}, \mathcal{R}[\sup_{x\in H} x]$.

Theorem 3.18 *Any relation $\mathcal{R}$ inducing a step-wise simulation between $\mathbb{G}_1$ and $\mathbb{G}_2$ also induce a simulation between $\mathbb{G}_1$ and $\mathbb{G}_2$.*

We will not include the proof of theorem 3.18 here. The essential proof steps are the following:

- Using the local transition-to-invariant transfer property, we explicitly transfer victory invariants for the original game ($\mathbb{G}_1$) to a product game following the structure of $\mathbb{G}_2$, but whose support is $G_1 \times G_2$. In that game, we allow the existential player to change state from $\mathbb{G}_1$ arbitrarily as long as it respects the order. We define the transferred invariant for product game by recovering the original game history from the product history, and using it to take decision relatively to $\mathbb{G}_2$.
- We then perform a minimization of the obtained victory invariant to be able to associate an unique history from $\mathbb{G}_2$ to any history of the product game
- Finally, we use the minimized victory invariant to derive a victory invariant in $\mathbb{G}_2$.

We then derive the correction of the iterative construction via a simulation from a suitable universal game, which lets the universal player change the iteration parameter x_A and the state `now` as it wishes as long as the state increase both with respect to source game order and with respect to the progression relation associated to the iteration construction. The body of the loop provides precisely the victory invariant we need for the simulation of regular steps, while the body of the divergence handler provides the victory invariant needed for limit steps. We do not include details here as the proof is very technical, notably because the intuitive idea of a game with product state does ensures chain-completeness. We need to embed that intuitive game in a completed variation.

Finally, we are left with recursive functions. In order for the introduction of recursive function to be correct, we only need to check that there exists victory invariants corresponding to the recursive function contract, as we can then readily apply induction hypothesis for the code that use it. According to our proof plan, we should obtain those victory invariants by reduction to iterations. We achieve that via the following ideas. First, we add extra transition to the game in a similar fashion as for continuations, so that it satisfy the contract of recursive calls. Second, we follows the victory invariant in that extended game. Whenever we should use a transition that does not exists in the original game, we simulate it by using the same method at the higher recursion level. Whenever that process generates an infinite cascade of such transition reductions, we match it to an infinite recursion stack and use the victory invariant from

divergence handler to finish one of those reduction. Note that again, that victory invariant may rely on transition reminiscent from recursive call as well, so it need to unfold them as well.

We note that our intuitive is compatible to the well-known technique of eliminating recursion with expliciting stacks, except that our stack can be transfinite. Note that in particular, this reduction will systematically reconstruct the implicit stack that we eliminated in the examples. Hence it is possible to see our technique as a systematic implicit compilation of recursion to iteration. We observe a gain because this compilation includes proofs, and as such is invisible in the final weakest pre-conditions, while the reduction proof is done once.

We now sketch the method that we use to encode recursion as non-returning iteration. Again, we do not provide full details as they are mostly technical.

- First and foremost, we fix the element that will be at the bottom of the stack.
- We use other constructions to transform the recursive structure in a different one that take non-empty sets of parameter/state pairs rather than regular parameter, totally ordered by the progression order. We also replace the progression order of the recurrence by the prefix order. Those totally ordered sets essentially summarize the stack trace. In order to be able to treat uniformly the function body and the divergence handler, we extends the post-condition of the function body so that it can choose to send a value corresponding to the post-condition of any level. This means that the divergence handler is equivalent to the function body restricted to limit chains. This also enforce that larger recursion level are associated to larger post-conditions.
- We inject a continuation call in the function body to empty the post-condition of the singleton stack, which takes care of the non-returning part.
- We choose a stack structure based on a well-founded non-empty sets of recursion levels, which are recursion parameters (which are themselves totally ordered sets at that point). We use a function to associate extra management information to each level, which corresponds naturally to data stored in activation frame. We notably store the actual parameter corresponding to the level.
- We associate to each level the victory invariant that we try to follow at this level, as well as the history corresponding to that invariant.
- We order stacks by a quasi-lexicographic order: larger stacks are bigger, and a stack is also bigger than another if they are equal until a level where the history increase. In order to enforce global growth of `now`, we need to restrict the order so that elements added to the increased history dominates the whole content of the smaller stack.
- We take as invariant the fact that the state has a valid corresponding stack, such that the current element is the last element of the history of the stack top element. A stack is valid if it satisfies the following conditions:

  (i) it is prefix-closed: if it contains a recursion level, which is a totally ordered set, then its contains all its non-empty prefix as well.

  (ii) History are strictly ordered: elements of any history in the stack are upper bounds of any history that is lower in the stack.

  (iii) history at any level admits a minimum.

  (iv) The victory invariant stored at a given level is a valid victory invariant in the game where we add transitions to satisfy all recursive calls at higher levels, for the singleton containing the minimum of the history and the post-condition corresponding to that level. Moreover, the associated history is in the invariant, and does not intersect the post-condition.

(v) The 'returns' are correctly linked: for any level and element y in the post-condition corresponding to that level, there exists a lower level in the stack such that adding y to the history of that level keep that history in the associated victory invariant.

(iv) For any level, if this level is a non-limit one, then the minimum of the associated history belongs to the pre-condition associated to that level.

- We create a valid singleton stack corresponding to the starting point of the recursion.

- We perform regular loop iteration by analyzing the transition requested by the top-level stack element. If it exists in the game, we perform it and create a larger stack by adding that element to top-level history, unless the post-condition is reached. In that case, we use condition (v) and well-foundedness to find the smallest level to which we can add the element, which create a larger stack. We need to take the minimum to make sure that element is not in the post-condition at this level. If the transition does not exists, we directly create a larger stack by adding a level for the corresponding recursive call. Note that condition (v) comes from the structure of game extended with 'recursive' transition.

- We perform loop divergence handling by noticing that from the structure of our order, only two cases can happen (highly non-trivial disjunction). First possibility, we can match the limit situation to a limit situation at a given constant level, with lower levels staying unchanged. In that case, we handle divergence by adding the least upper bound of history at that level, from step (iii) of victory invariants. If it happens to be in the post-condition, we then perform the same reduction as for regular steps. Second possibility, we can match the limit situation to a limit transfinite sequence of growing stacks, with levels staying unchanged. In that case, we can use condition (iv) to match it to a limit situation of our original recurrence. In particular, those corresponds to recursive calls to limit levels, so we handle them by adding a level for the corresponding recursive call. Again, if the post-condition happens to be true immediately, we then perform the same reduction as for regular steps.

3.6 Relative Completeness

For verification, we only need our proof methodology to be sound. However, it is also relatively complete, in the sense that if we can prove directly the existence of a victory invariant for a definable set pair in the meta-logic, then we can also prove its existence by finding a program of $\mathcal{W}_G$ which satisfy the corresponding contract, via corollary 3.11. The proof amounts to create a program that follows the explicit victory invariant. We make a few implicit hypothesis about the meta-logic for the proof to work out, notably that it can talk about victory invariants of the considered game, and express the notion in a manner compatible with our definition. We also need to be able to express a few set operators.

Lemma 3.19 *For all games $\mathbb{G} = (G, \preccurlyeq, \Delta)$, all well-typed formulas pair φ_0, φ_1 not mentioning* `old`*, and all assignations σ, if the existence of a victory invariant for the set pair $\{x \in G \mid [\![\varphi_0]\!]_{\sigma,\mathsf{now}\leftarrow x}\}, \{x \in G \mid [\![\varphi_1]\!]_{\sigma,\mathsf{now}\leftarrow x}\}$ is provable in the meta-logic, then there exists a well-typed program π with output type $\{0\}$ such that $\mathcal{C}[(\Gamma_{\mathbb{G}})_{x\in G}, \sigma]\left(\pi : \langle \varphi_0 \hookrightarrow u_{\{0\}}.\ \varphi_1\rangle\right)$ holds.*

Proof: We exhibit the corresponding program as

$$
\begin{array}{l}
\mathsf{switch} \\
\mathsf{case}\ i_{\mathcal{P}(\mathcal{P}(G))} : \varphi_{INV}\ \mathsf{in} \\
\quad \mathsf{rec}\ f^{\{0\}}_{\mathcal{P}(G)}(h_{\mathcal{P}(G)}) : \left\langle \begin{array}{l} h_{\mathcal{P}(G)} \in i_{\mathcal{P}(\mathcal{P}(G))} \wedge \mathsf{now} = \max h_{\mathcal{P}(G)} \\ \wedge\ \forall x.\ x \in h_{\mathcal{P}(G)} \wedge \varphi_1[\mathsf{now} \leftarrow x] \Rightarrow x = \mathsf{now} \end{array} \hookrightarrow u_{\{0\}}.\ \varphi_1 \right\rangle \\
\qquad \mathsf{progress}(\sqsubset, h_{\mathcal{P}(G)}) \\
\qquad = \mathsf{switch} \\
\qquad\quad \mathsf{case}\ u_{\{0\}} : \varphi_1\ \mathsf{in}\ (0)_{\{0\}} \\
\qquad\quad \mathsf{case}\ u_{\{0\}} : \neg\varphi_1\ \mathsf{in\ let}\ u_{\{0\}} = \mathit{step}^{\{0\}}_{\mathcal{P}(G)}\ \mathsf{in}\ f^{\{0\}}_{\mathcal{P}(G)}(h_{\mathcal{P}(G)} \cup \{\mathsf{now}\}) \\
\qquad\quad \mathsf{end\text{-}switch} \\
\qquad \mathsf{catch}(\mathcal{H}_{\mathcal{P}(\mathcal{P}(G))}, v_{\mathcal{P}(G)^{\mathcal{P}}(G)}, w_{\mathcal{P}(G)^{\mathcal{P}(G)}}) : \\
\qquad\quad f^{\{0\}}_{\mathcal{P}(G)}(\bigcup H_{\mathcal{P}(\mathcal{P}(G))} \cup \{\mathsf{now}\}) \\
\quad \mathsf{in}\ f^{\{0\}}_{\mathcal{P}(G)}(\{\mathsf{now}\}) \\
\mathsf{end\text{-}switch}
\end{array}
$$

where none of the immutable variables bound by the program occurs in φ_1, φ_{INV} expresses the fact the $i_{\mathcal{P}(\mathcal{P}(G))}$ is a victory invariant for the set defined by φ_0 and φ_1 relatively to $\mathbb{G}$, and $\sqsubset$ is the strict prefix order. □

4 Linking Games and Transition Systems

In that section we show how to connect transition systems and games, in order to recover more intuitive transfer properties. Indeed, our game-based framework do not represent directly the typical small-steps operational semantics of programming language. We connect the two notions by presenting several closely related translations from transition systems with observation records to games. The details of the translations differ depending on the actual transfer property. We show here how to obtain three different transfer results: program correctness, existence of behaviors, and simulation.

We first define the notion of transition systems with observation records. We use transition systems because they are the natural abstraction for small-step operational semantics. We equip them with observation records to model abstractly the trace of observable effects of the program, which are the natural tools to describe accurately the behavior of a non-terminating programs.

Definition 4.1 (Transition System with Observation Records) *A* transition system with observation records *is a quintuplet $\mathcal{S} = (S, \rightarrow, p, O \preccurlyeq)$ where*

- *S is the domain of the transition system, and $\rightarrow \subseteq S \times S$ the transition relation*
- *$(O, \preccurlyeq)$ is a chain-complete order, representing the observation records*
- *p is a map from S to O, which extract the observation record from a state*

From the definition above, we use the least upper bound of the observation records to abstract infinite executions. Note that if we consider everything as observable, we can immediately translate any transition system to a transition system with observation records by taking a domain made of non-empty finite state sequences, and observation records made of potentially infinite state sequences. We can similarly translate labeled transition system with observable labels by adding the sequence of observed labels to the state, and use those sequence as observation records.

We now define the two translations corresponding respectively to the angelic (existential) and demonic (universal) interpretation of non-determinism.

Definition 4.2 (Timed support) *The* timed support *associated to a transition system with observation records $\mathcal{S} = (S, \to, p, O, \preccurlyeq)$ is the ordered set $(G_{\mathcal{S}}, \preccurlyeq_{\mathcal{S}})$ defined by*

- $G_{\mathcal{S}} = \{(n, x) \in (\mathbb{N} \cup \{\infty\}) \times (S \cup O) \mid (n = \infty \wedge x \in O) \vee (n \neq \infty \wedge x \in S)\}$
- $\forall n \in \mathbb{N}, x \in S, y \in O.\ (n, x) \preccurlyeq_{\mathcal{S}} (\infty, y) \Leftrightarrow p(x) \preccurlyeq y$
- $\forall n, m \in \mathbb{N}, x, y \in S.\ (n, x) \prec (m, y) \Leftrightarrow n < m \wedge p(x) \prec p(y)$
- $\forall x, y \in O.\ (\infty, x) \preccurlyeq_{\mathcal{S}} (\infty, y) \Leftrightarrow x \preccurlyeq y$

Definition 4.3 (Existential game) *The* existential game *associated to a transition system with observation records $\mathcal{S} = (S, \to, p, O, \preccurlyeq)$ is defined as the game $\mathcal{G}_{\mathcal{S},\exists} = (G_{\mathcal{S}}, \preccurlyeq_{\mathcal{S}}, \Delta_{\mathcal{S},\exists})$ with transitions defined by*

$$\Delta_{\mathcal{S},\exists}((n, x)) = \begin{cases} \{\{(n+1, y)\} \mid x \to y\} & n \in \mathbb{N} \\ \emptyset & n = \infty \end{cases}$$

Definition 4.4 (Universal game) *The* universal game *associated to a transition system with observation records $\mathcal{S} = (S, \to, p, O, \preccurlyeq)$ is defined as the game $\mathcal{G}_{\mathcal{S},\forall} = (G_{\mathcal{S}}, \preccurlyeq_{\mathcal{S}}, \Delta_{\mathcal{S},\forall})$ with transitions defined by*

$$\Delta_{\mathcal{S},\forall}((n, x)) = \begin{cases} \{\{(n+1, y) \mid x \to y\}\} - \{\emptyset\} & n \in \mathbb{N} \\ \emptyset & n = \infty \end{cases}$$

Those definitions are valid only because the timed support is chain-complete, which is a routine consequence of chain-completeness of observations. Note that the removal of the $\emptyset$ from the universal transition is not an artificial constraints but stems from intuition. Indeed, a state with no possible transition corresponds to a stuck state, from which we should not be able to progress. Moreover, this ensures that the definition of the universal game and of the existential game coincide when the transition system is deterministic. This matches the intuition that the interpretation of non-determinism has no influence for that particular case.

We now prove two lemmas that provides transfer properties for transition systems. The first relate existence of victory invariants in existential games to existence of behaviors for transition systems, while the second relate existence of victory invariants in universal games to eventual (infinite) reachability in the transition system. This second notion matches naturally Hoare-style correctness properties.

Lemma 4.5 *For all transition systems with observation records $\mathcal{S} = (S, \to, p, O, \preccurlyeq)$, for all sets $P, Q \subseteq S, Q_\infty \subseteq O$, there is a victory invariant relative to $\mathbb{G}_{\mathcal{S},\exists}$ for $\mathbb{N} \times P, \mathbb{N} \times Q \cup \{\infty\} \cup Q_\infty$ if and only if for all element $s_0 \in P$, either*

(i) there exists a finite transition sequence $s_0 \to \ldots \to s_n$ of $\mathcal{S}$ such that $s_n \in Q$

(ii) there exists an infinite sequence $s_0 \to \ldots \to s_n \to \ldots$ of $\mathcal{S}$ such that the least upper bound of observation records over the sequence belongs to Q_∞

Proof: The reciprocal direction is a direct consequence of the transfer property for games, using the same style of program as for the completeness lemma 3.19. We build a program that first exhibits the trace with a switching statement, then follows it until its potentially infinite end using a tail-recursive function, with a divergence handler for the infinite case.

For the direct case, we can actually use a similar technique. We first build the same program than for the completeness lemma 3.19, which is correct because a victory invariant exists. Then, we add an extra parameter to the tail-recursive function which logs the actual sequence of transition, and enforce its growth via the progression clause. We can achieve it by replacing the parameter type by a product. It is routine to check that this preserve the correctness of the program. However, we can now prove

a seemingly stronger post-condition which provides exactly the existence of such a trace. Since the existential game does not provide transitions to the empty set, we can build an history that contains an element satisfying the post-condition, hence showing that the desired trace exists. The simplest way to build such an history is to take a maximal one, which exists via Zorn's lemma. □

Lemma 4.6 *For all transition systems with observation records $\mathcal{S} = (S, \to, p, O, \preccurlyeq)$, for all sets $P, Q \subseteq S, Q_\infty \subseteq O$, there is a victory invariant relative to $\mathbb{G}_{\mathcal{S},\forall}$ for $\mathcal{N} \times P, \mathbb{N} \times Q \cup \{\infty\} \times Q_\infty$ if and only if for all potentially infinite transition sequences $s_0 \to \ldots \to s_n(\to \ldots)^?$ of $\mathcal{S}$ such that $s_0 \in P$, either*

(i) there exists an element s_i in the sequence such that $s_i \in Q$

(ii) the sequence is infinite, and the least upper bound of observation records along the sequence is in Q_∞

(iii) the sequence is finite with last element s_n, and there exists $x \in S$ such that $s_n \to x$

Proof: For the reciprocal direction, we can directly use the transfer property for games. Indeed, the tail-recursive program that steps until Q is reached, or until divergence, will give the desired victory invariant. To that end, we take as argument the finite sequence of states processed until there, with the pre-condition that Q is not in the sequence. The condition (iii) then guarantee that we can perform a step, from which we either reach $\mathbb{N} \times Q$ or make a recursive call. If the recursive function diverges, the state is the least upper bound of the corresponding infinite sequence, from which we derive $\{\infty\} \times Q_\infty$.

For the direct case, we cannot directly use the program logic to derive the result, as we did for the existential game. We first pre-process the universal game so that transition follows the fixed transition sequence. Consider a potentially infinite transition sequence $s_0 \to \ldots \to s_n(\to \ldots)^?$. We define an intermediate game $\mathbb{G}$ which is the universal game, except that for each state s_i which is not the last of the sequence, the transition from (i, s_i) is restricted to $\{\{(i+1, s_{i+1}\}\} \subseteq \Delta_{\mathcal{S},\forall}((i, s_i))$. In particular, it is immediate to check that the victory invariant we have for the universal game is also valid for $\mathbb{G}$. Then, following the spirit of the proof for the existential game, we build the same program than for the completeness lemma 3.19, but with respect to $\mathbb{G}$ instead. We then add an extra argument to the tail-recursive function which logs the actual sequence of transition, and again enforce its growth via the progression clause. Using the pre-condition, we enforce that this sequence is comparable with our original sequence for the prefix order, as a consequence of the restriction of transitions. In particular, we can enforce as final post-condition that the state is the least upper bound of some sequence which is comparable with the original transition sequence for the prefix order. Since the universal game does not contain transition to the empty set, we then recover the existence of such sequence/state pair. At this point, the following cases may arise:

- The sequence constructed by the program is strictly bigger than the original transition sequence. In that case, condition (iii) holds for the original transition sequence.
- The sequence constructed by the program is strictly shorter than the original transition sequence, or finite and equal. In that case, condition (i) holds for the original transition sequence.
- Both sequences are infinite. That case corresponds to condition (ii).

□

Using a product construction, we now derive a third translation, corresponding naturally to simulation.

Definition 4.7 *Given two transition systems with observation records $\mathcal{S}_1, \mathcal{S}_2$, the simulation game from $\mathcal{S}_1$ to $\mathcal{S}_2$ is the game $\mathbb{G}_{\mathcal{S}_1 \to \mathcal{S}_2} = (G_{\mathcal{S}_1} \times G_{\mathcal{S}_2}, \preccurlyeq_{\mathcal{S}_1} \times \preccurlyeq_{\mathcal{S}_2}, \Delta_{\mathcal{S}_1 \to \mathcal{S}_2})$ with transitions defined by:*

$$\Delta_{\mathcal{S}_1 \to \mathcal{S}_2}((x, y)) = \{\{x\} \times Y \mid Y \in \Delta_{\mathcal{S}_2,\exists}(y)\} \cup \{X \times \{y\} \mid X \in \Delta_{\mathcal{S}_1,\forall}(x)\}$$

Lemma 4.8 *For all transition systems $\mathcal{S}_1 = (S_1, \rightarrow_1, p_1, O_1, \preccurlyeq_1)$ and $\mathcal{S}_2 = (S_2, \rightarrow_2, p_2, O_2, \preccurlyeq_2)$, for all set $P \subseteq S_1 \times S_2$, set family $(Q_{a,b})_{a,b\in\mathbb{B}}$ such that $Q_{a,b} \subseteq (a?S_1 : O_1) \times (b?S_2 : O_2)$, there exists a victory invariant for the set pair made of the pre-condition $\mathbb{N} \times P$ and of the post-condition*

$$\{((n, x), (m, y)) \in G_{\mathcal{S}_1} \times G_{\mathcal{S}_2} \mid (x, y) \in Q_{n=\infty,m=\infty}\}$$

if and only if for all $t_0 \in S_2$, for all maximal transition sequence $s_0 \rightarrow \ldots \rightarrow s_n(\rightarrow \ldots)^?$ of $\mathcal{S}_1$ such that $(s_0, t_0) \in P$, there exists a transition sequence $t_0 \rightarrow \ldots \rightarrow t_n(\rightarrow \ldots)^?$ of $\mathcal{S}_2$ such that:

(i) the transition sequence from $\mathcal{S}_2$ is finite with maximum t_n, and there exists i such that $(s_i, t_i) \in Q_{false,false}$

(ii) the transition sequence from $\mathcal{S}_2$ is finite with maximum t_n, the transition sequence from $mathcalS_1$ is infinite with least upper bound of observation s_∞, and $(s_\infty, t_n) \in Q_{true,false}$

(iii) the transition sequence from $\mathcal{S}_2$ is infinite with least upper bound of observation t_∞, and there exists i such that $(s_i, t_\infty) \in Q_{false,true}$

(iv) the transition sequences are both infinite with respective least upper bound of observations s_∞, t_∞, and $(s_\infty, t_\infty) \in Q_{true,true}$

Proof: The proof of this lemma is essentially an hybrid of proofs of lemma 4.5 and 4.6.

First note that the maximality condition amounts to rule out the condition (iii) from the universal lemma 4.6. We can then obtain a program for the reciprocal direction as follows. First, we use a tail-recursive function to create a sequence such that one of the condition (i), (ii), (iii) or (iv) holds when using the least upper bound of the sequence as the witness s_i or s_∞. Since such an elements exists for maximal sequence, we can always step if we have not found such an element yet. Note that this phase use only the step function coming from the universal side of the game. Once such a sequence is found, we use the switching construction to build all the existential witness, then we build a tail-recursive function that goes to the right element by following the obtained trace. Note that this second phase use only the step function coming from the existential side of the game.

For the direct case, the proof is also an hybrid. First, we perform a pre-processing similar to the one done for the proof of the universal lemma in order to fix the game transition to the ones corresponding to our maximal transition sequence. Then, we build the same program than for the completeness lemma 3.19, and add two extra recursive parameters for the sequence of states for each transition systems. Following the reasoning from lemmas 4.5 and 4.6, we obtain a post-condition asserting the existence of two sequences with least upper bounds related to the final state pair: a sequence for $\mathcal{S}_1$ which is prefix of the fixed maximal sequence, the converse case being ruled out here by maximality, and an arbitrary sequence for $\mathcal{S}_2$. The diverse possibilities then maps exactly to conditions (i)-(iv).

□

5 Related Work and Future Work

Hoare logic, games and dual non-determinism Games were introduced as a model for weakest pre-condition transformers by Back and von Wright [1]. A related Hoare logic is studied by Mamouras [6]. In both cases, the focus of those works is to model and verify programming language containing dual non-determinism. Our work differs in the sense that games are used as a model of small-step semantics, and that the nature of non-determinism is only chosen to obtain the transfer property we are interested in. In particular, we may obtain games with both types of non-determinism only when considering relational properties, or internally as our proof of soundness construct such hybrid games for the recursive case.

Hoare logic for machine code in proof assistants Hoare logics defined for machine code in proof assistants are typically defined by proof rules which are mostly unrelated to the syntax of the underlying program. Such logics are closely related to our work, as proof rules match the syntax cases of auxiliary programs. Moreover, automation by tactics may simulate a weakest pre-condition calculus. A typical example is the work of Myreen [8], which corresponds to a `While` language extended by well-founded recursion for the auxiliary language. Note that well-founded recursion is inherently tied to the meta-logic induction. In contrast, we support non-terminating recursion and proof of diverging behaviors. Note that step-indexed logics like Iris [4] support non-terminating recursion by the mean of the Löb rule, as well as arbitrary higher-order programs, but drop any support for total correctness, while we support fine-grained specification/proof for non-terminating recursion but limited to first-order.

Verified compilation by the mean of program logics In Section 2.3, we sketched how our approach could be used to verify a compiler. The idea of using program logic to ease compiler proof is not new, and has been notably used for the proof of the CakeML certified compiler [5]. However, a notable difference with our work is that the logic is used to prove existence of behaviors for the target code, while we can prove simulation directly, in any direction by swapping the role of languages for the transfer property. Moreover, the proof rely on the determinism of the target language, while our methodology support non-determinism for both the source and the target language. Finally, in order to prove existence of non-terminating behaviors, the proof of CakeML relies on Hoare triples with an unusual interpretation in temporal logic, which means that properties about non-terminating behaviors are only stated in terms of its finite prefixes. On the other hand, our approach take the position of talking directly about the result of infinite executions.

Our approach can be extended in several directions. A first direction would be toward separation logic. We believe that adding a separation logic layer a posteriori in a similar fashion as the first-order fragment of Iris [4] should pose no trouble. Another direction would be to generalize our approach to an higher-order settings. This can be achieved reasonably if we enforce a restriction of the higher-order: either by limiting the rank of the allowed functions, or by limiting the usage of recursive calls so that they do not occur under closure themselves passed to recursive calls. We do not know if it is possible to achieve the best of both worlds, which would remove both limitations at once. A last possible direction would be to add concurrency. While our approach obviously support sequential consistency, it is not at all evident that the support is nice enough for practical usage. Indeed, our auxiliary language have no support for parallel program composition.

References

[1] R. J. R. Back and J. von Wright. Duality in specification languages: a lattice-theoretical approach. *Acta Informatica*, 27(7):583–625, Jul 1990.

[2] Ralph-Johan Back and Joakim von Wright. *Refinement calculus - a systematic introduction*. Undergraduate texts in computer science. Springer, 1999.

[3] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18:453–457, August 1975.

[4] Robbert Krebbers, Ralf Jung, Aleš Bizjak, Jacques-Henri Jourdan, Derek Dreyer, and Lars Birkedal. The essence of higher-order concurrent separation logic. In *Proceedings of the 26th European Symposium on Programming Languages and Systems - Volume 10201*, pages 696–723, New York, NY, USA, 2017. Springer-Verlag New York, Inc.

[5] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 179–191, New York, NY, USA, 2014. ACM.

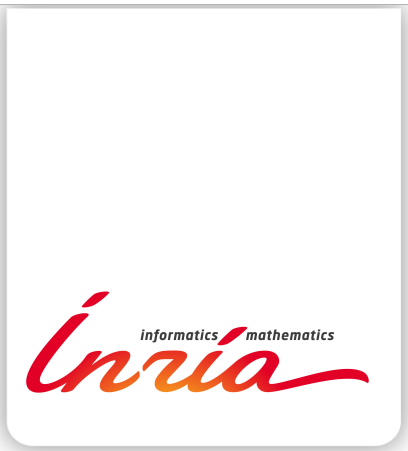
[6] Konstantinos Mamouras. Synthesis of strategies using the hoare logic of angelic and demonic nondeterminism. *Logical Methods in Computer Science*, 12(3), 2016.

[7] Z. Manna and J. McCarthy. Properties of programs and partial function logic. In *Machine Intelligence*, volume 5, 1970.

[8] Magnus O. Myreen and Michael J. C. Gordon. Hoare logic for realistically modelled machine code. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 568–582, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[9] Keiko Nakata and Tarmo Uustalu. A hoare logic for the coinductive trace-based big-step semantics of while. In Andrew D. Gordon, editor, *Programming Languages and Systems*, pages 488–506, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[10] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Communications of the ACM*, 10:501–506, 1967.



RESEARCH CENTRE
SACLAY – ÎLE-DE-FRANCE

1 rue Honoré d'Estienne d'Orves
Bâtiment Alan Turing
Campus de l'École Polytechnique
91120 Palaiseau

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399