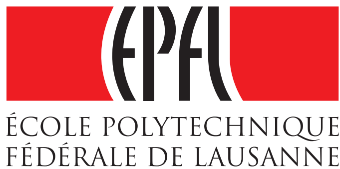


Master Thesis Project

Termination Analysis in a Higher-Order Functional Context

Nicolas Voirol

EPFL
School of Computer and Communication Sciences

Fall 2013

Supervisor	**Professor**
Etienne Kneuss	Prof. Viktor Kuncak
EPFL	EPFL

Abstract

We start by describing how higher-order function support can be added to a (first order) functional verification framework. We cover both the higher-order construct management and framework extensions necessary for constraint specification. Next, we outline a termination proof constructor for the aforementioned framework. In order to provide effectiveness as well as performance, our approach combines different techniques in a *weeding-out* strategy where each partial proof is carried on to the next prover. To enable high-level reasoning about formulas during termination proofs, we make use of the underlying verification framework while maintaining soundness by carefully restricting the acceptable formula space. Finally, we propose an extension to the framework input language that adds support for generic type polymorphism while maintaining compatibility with legacy features.

1 Introduction

Higher-order functions are ubiquitous in functional programming as they provide a powerful and elegant abstraction to the developer. Any verification framework aiming for real-world program analysis must therefore support the use of some kind of higher-order construct. These are provided through the addition of function types, function applications (distinct here from method calls) and anonymous functions.

The key role of a verification framework consists in proving certain properties on a program, sometimes given certain assumptions. In order to state meaningful properties and assumptions on higher-order functions, we need some means of specification. There are different options for such requirements description (type based [6, 14], closure based, etc), and we present a restricted variant of universal quantification that fits well with the pre-existing framework.

The actual verification relies on a pattern based instantiation algorithm that generates constraints on function applications. These constraints are introduced by a flexible system that enables handling of complex multivariate requirements. Anonymous functions are maintained (with closures) by our system until application sites where their body is inlined into the formula, thus reducing the impact on the underlying framework.

Termination proving, notwithstanding the halting problem, can be tractable and even efficient in many concrete programs, making it a worthy addition to any verification framework, even more so in our case as the underlying framework is complete *given termination*. This presents interesting challenges during proof construction as we use the verifying capabilities of the system to provide a powerful formula abstraction during reasoning.

We present a modular approach (derived from [5]) where *termination problems* are broken up into independent sub-components which are treated individually. The result of each technique application then consists in a subset of the input problem where only harder parts of the problem remain to be proven. We encode our basic termination problems as sets of methods which are passed from one technique to another. These problems can also be extended with additional information computed during (failed) proof construction, which can be used further along by a more powerful technique.

A multitude of techniques can then be injected into this modular scheme. We focused mainly on size-based termination [10], but most of the principles can be extended to numeric convergence. We will discuss three such techniques in the scope of this paper.

Generic type polymorphism provides a flexible and powerful means to reuse type-independent constructs (which is the case of most algebraic data types). Moreover, any framework that aims to support real-world applications should support these to some extend as they are present in many – if not most – modern languages.

The SMT solver that powers most of the proofs in the underlying framework does not support generic algebraic data types, so this feature needs to be simulated by our extension for any possibility of generic verification. We present a type unfolding algorithm that complements the unrolling nature of the framework proof construction.

2 Higher-Order Functions

We trivially extend the Leon syntax (*ie.* PureScala [2]) with higher-order constructs by following the Scala syntax (see Figure 2.1).

$$\begin{array}{rcl} \textit{type} & ::= & \textit{PureScala type} \\ & & \mid \textit{type} \;\Rightarrow\; \textit{type} \\ \textit{expr} & ::= & \textit{PureScala expr} \\ & & \mid (\ \textit{decls}\ ) \;\Rightarrow\; \textit{expr} \\ & & \mid \textit{expr}\ (\ \langle\ \textit{expr}\ \langle\ ,\ \textit{expr}\ \rangle^*\rangle?\ ) \\ & & \mid \textbf{forall}\ (\ (\ \textit{decls}\ ) \;\Rightarrow\; \textit{expr}\ ) \end{array}$$

Figure 2.1: Syntax extensions for higher-order functions support

2.1 Requirements specification

A natural abstraction for functions are relations, along with the additional constraint that, for a relation R,

$$\forall (r_1, r_1'), (r_2, r_2') \in R.\, r_1 = r_2 \implies r_1' = r_2'.$$

Moreover, powerful tools for stating properties on relations (as we can see in the above formula) are universal and existential quantification. In the context of deterministic programs, it is generally more useful to consider universal than existential quantification, so we dismissed the later in our extension.

Bearing these considerations in mind, we extend the framework syntax with a **forall** construct. This statement expresses universal quantification with the restrictions that 1) the **forall** statement must be a top-level conjunct in the requirement, to disallow existential quantification expressed with inner and outer negation, 2) **forall** statements *must* constrain function applications (and only function applications), to avoid unsupported uses, 3) one cannot define *recursive* constraints with the **forall** construct as in $\forall a. f(a{+}1) > f(a)$, since the implications of such constraints cannot be captured by pattern instantiation, and 4) the arguments to function applications can only contain surjective operators, as equalities with concrete values could prove meaningless otherwise. A generalized **forall** statement would be an interesting extension in its own right but is out of the scope of this paper. Note that conditions 3) and 4) are satisfied in our case by requiring these arguments to either a) *not* include universally quantified variables, or b) consist *only* of a quantified variable (*eg.* $f(a + 1)$ is in fact disallowed).

One could criticize the simplicity of this construct in favor of a more complex model that exposes memory structures before and after execution (see [11] for a more complete

example), but such approaches do not scale well and lack clarity. Universally quantified requirements offer a flexible means of specification without introducing unnecessary convolutions.

2.2 Generating constraints

The constraint generation algorithm is a particular case of pattern based universal quantification as implemented in modern SMT solvers. Directly using Z3 [4] universal quantification unfortunately did not pan out (presumably) because of model completeness requirements. Our system provides this functionality in a flexible way that supports the method invocation unfolding scheme implemented by the underlying framework [13].

2.2.1 Pattern based quantification

Universal quantification can be viewed as a pattern generalization, where universally quantified variables act as named wild-cards. Expressions that match the pattern given wildcard instantiation are then constrained by the quantified proposition. We call this process *instantiating* the pattern.

Our case is slightly different as we only consider universal quantification in the context of higher-order function constraining. This means that pattern matching is a much simpler task than in the general setting. Indeed, it suffices to perform matching on higher-order function applications in pattern and expression since other types of wild-card patterns are explicitly disallowed. Furthermore, we require parse tree equality for higher-order functions due to the lack of formalism of true function equality (undecidable). It may be interesting to relax this constraint in future work by using a type-based matching to allow a wider range of universally quantified specifications .

2.2.2 Defining patterns

The main observation behind our pattern definition is that the entry requirement (*precondition*) holds at all times once inside the body of a method. This means that while visiting the body, we are guaranteed that the *precondition* can be asserted at any given point without constraining a possible solution any further. Furthermore, since the whole *precondition* expression holds, any sub-proposition of a top-level conjunction must hold as well, and can also be asserted anytime during body traversal.

These considerations, associated with the description of our restricted universal quantification, lead to the following natural definition: a *pattern* is a conjunct of a universally quantified proposition that contains universally quantified variables as well as constrained higher-order function applications. Note that from 2.1, universally quantified propositions *must* be top-level conjuncts in the *precondition* expression. For example, if the *precondition*

of a function definition is stated as

$$f(1) = 1 \land \forall b.g(b) \neq 0 \land (\forall a.f(a) < 0 \lor f(a) > 0),$$

we can extract the two patterns $\forall b.g(b) \neq 0$ and $\forall a.f(a) < 0 \lor f(a) > 0$.

2.2.3 Constraints from context

As presented in section 2.2.1, pattern matching is performed via higher-order function application comparison. This means that to match a pattern, we must have seen concrete applications for all higher-order functions that are applied in the pattern. At most one function application can be encountered by a visitor at any given time and interacting program components are not necessarily close to each other, so local search is not sufficient. We therefore need a global mechanism to accumulate visited applications until patterns can be instantiated. In order to keep track of previous applications, we define the concept of *context*, a partial map from application callers to sets of concrete applications. This map is used to check whether patterns can be matched by verifying that callers in patterns all have at least one concrete counter-part in the context.

Whenever an application is discovered by the visitor, it is used in conjunction with the current context to instantiate new constraints. Each pattern is considered, and function applications contained in the pattern are associated to all possible combinations of concrete applications afforded by the context. These associations are then fed to the pattern instantiation system which generates a set of new constraints. Note that only combinations featuring the newly discovered application need be instantiated, as others have already been handled (when they were discovered). Finally, the new application is added to the context for subsequent instantiations.

2.2.4 Pattern instantiation

We mentioned in section 2.2.1 that patterns are *instantiated*. Simply asserting the pattern is meaningless, as the universally quantified variables do not capture as such the actual values figuring in the concrete expression. We need a more complex behavior that links patterns and concrete applications. As we are guaranteed by 2.1 that universally quantified variables in application arguments can only appear as variable arguments, we could solve this issue by building a mapping from the quantified variables to concrete expressions. This mapping associated with equality constraints for variables that have multiple images would provide the necessary binding, but we present a different option that fits better with a possible relaxation in future work of the constraint on the shape of universally quantified application arguments.

We propose a solution where the binding is enforced by equality constraints between pattern and concrete arguments expressions (note that this poses no restriction on the pattern shape). If we consider the pattern $f(a, 1) > 0$ and the concrete application $f(2, 1)$,

we would assert $f(a, 1) > 0$ and generate the additional constraints $a = 2$ and $1 = 1$. These new equality constraints are not so trivial to manage:

- They cannot be asserted (or analogously linked by conjunction with the pattern). Indeed, patterns like $f(a, 1) = 1$ would break satisfiability of any program calling $f(x, 2)$ ($1 = 2 \notin SAT$).
- They cannot imply the pattern. The counter-example driven nature of the underlying verification framework will fulfill implication truth requirement with $false \implies *$ by setting free variables to arbitrary values, thus leaving concrete applications unconstrained to generate counter-examples.

We solve this issue by separating equality constraints into two sets: *hard constraints* and *soft constraints*. Hard constraints are used to bind free variables by asserting equality constraints containing new free variables, whereas soft constraints provide the flexibility to handle constraints that can *never* be satisfied by using implication:

$$hard \wedge (soft \implies pattern)$$

In order to compute the collection of hard constraints, we start by assigning each constraint to the set of free variables contained within it. Given these sets, we then group constraints and order the groups by increasing free variable set size. Finally, we traverse the sequence of groups and select one constraint per group and add it to the hard constraints. We must make sure to select the constraint with least structural size to avoid underconstraining caused by surjectivity (*eg.* $g(a) = g(x)$ weaker than $a = x$ but contains the same free variables). All remaining constraints are soft constraints.

Since higher-order function equality is not allowed in our system, equality constraints as described above do not extend to universally quantified higher-order variables. Happily, due to the lack of operators on functions, these can be elegantly handled via the mapping option we presented previously without constraining argument shape. We therefore collect all equality constraints inducing a mapping from universally quantified function-typed variables to concrete expressions (of the shape $f_1 = expr$ where f_1 is universally quantified and of function type). This mapping is then used to replace all other instances of these variables by concrete expressions (*eg.* $f_1(x)$ becomes $expr(x)$). As this replacement may create new concrete applications, these must also be constrained by running the complete algorithm over the generated constraint.

2.2.5 Inductive invariants

We have discussed how to generate constraints based on universal quantification in the precondition, but we have not yet addressed constraints that appear through inductive invariants (*ie.* postcondition assumption).

Postcondition assumption is handled like preconditions in the general case. Indeed, we know the postcondition must be true during assumption, so we can instantiate the universally quantified propositions contained within it as patterns based on method invocation arguments. We use the same context as for preconditions, extended with a partial mapping from methods to concrete invocations (to encode the invocation result). The same procedure for argument equality is used as for variable function applications.

2.3 Universal quantification in formulas

We have discussed pattern based instantiation for universally quantified propositions found in the precondition, but the driving observation behind precondition pattern validity does not hold in general. By definition, universal quantification can only appear in the pre- or postcondition, and and we know from [13] that these are either handled as presented previously, or appear on the right-hand side of a top-level implication in the formula.

Since the framework is counter-example based, we know it will be searching to satisfy these formulas with a model m such that $m \models \neg(a \implies b)$, thus disproving the base proposition. This implies that $m \models a$ and $m \models \neg b$. By definition of our restricted universal quantification, all universally quantified propositions are top-level conjuncts in b. Hence, there must be at least one universally quantified proposition that does not hold in b. In other words, we can replace all patterns of the shape $\forall \bar{a}'_i.q'_i$ by $\exists \bar{a}'_i.\neg q'_i$ in b while maintaining full equivalence. Since our formula is then fed to a SMT solver that will search for a model m, existential quantification is assumed by default for free variables in the formula. This means we can drop the explicit existential quantification from b as long as we make sure all $\bar{a}_i$ are free over the whole formula.

Sometimes, preconditions located on the right-hand side of the implication are issued from method invocations that take place in an anonymous function body. In this case, we generally do not yet have the concrete arguments to the anonymous function application, so the formula will contain free variables (*ie.* the anonymous function formal arguments). As we wish our proofs to remain general and not depend on eventual future invocations, we simply ignore whether or not we are in an anonymous function during proof construction. This makes the proof harder to build, but it also makes it sufficiently general to satisfy any concrete arguments.

2.4 Formalization

In this section, we formally describe the procedure that quite naturally emerges from the requirements specification and constraint generation schemes presented above.

We start by defining the two classes of formulas we are considering. Let $\mathcal{L}^{\Pi}$ be the *base theory* (logic) already supported by the verification framework [13]. We also borrow symbols program Π, implementation $impl^{\Pi}_f$, as well as pre- and postcondition $prec^{\Pi}_f$ and $post^{\Pi}_f$ from [13]. Let $\mathcal{L}^{\Pi\mathcal{U}}$ be the extension of $\mathcal{L}^{\Pi}$ that supports the restricted case of universal

quantification described in 2.1 and function variables (these can actually be handled in $\mathcal{L}^{\Pi}$ by translating them to maps from argument tuples to results [12]).

By definition, the universal quantification construct we introduced in $\mathcal{L}^{\Pi\mathcal{U}}$ can *only* be found in top-level conjuncts of $prec_f^{\Pi}$ and/or $post_f^{\Pi}$. We can therefore infer that these formulas (given some trivial rearrangements) will be of the shape $(\bigwedge_i \forall\bar{a}_i.q_i) \wedge p$ where $p \in \mathcal{L}^{\Pi}$ (modulo function variables as maps, which will be the general assumption from here on), and can be rewritten as $(\bigwedge_j \forall\bar{a}'_j.q'_j) \wedge p'$ where for all j, q'_j is *not* a conjunction and $p' \in L^{\Pi}$. Note that each conjunct $\forall\bar{a}'_j.q'_j$ represents a pattern as defined in 2.2.2. Finally, we define $\rho\,(expr)$ as the formula encoding of the context-flow-graph to *expr* in the current definition (*ie.* path to *expr*).

Algorithm 1 Pseudo-code for constraint generation procedure with higher-order functions

precondition := $prec_f^{\Pi}$
procedure constrain(q)
  formula := $q\left[\lfloor prec_f^{\Pi}\rfloor, post_f^{\Pi}/E, prec_g^{\Pi}/E\right]$
  for $f_i(\bar{x})_i$ **in** formula
    for $\forall\bar{a}'_j.q'_j$ **in** precondition
      constraints := $\bigwedge$(context, $f_i(\bar{x})_i) \models \forall\bar{a}'_j.q'_j$
      formula := formula $\wedge\ \rho\,(f_i(x)_i) \implies$ constraints
    done
    context := context $\cup\ \{f_i(\bar{x})_i\}$
  done
  return formula
end

The constraint procedure in Algorithm 1 is applied to all formulas generated by the underlying framework. Then, to integrate with invocation unrolling, it is applied to unrolled invocation bodies whenever these are introduced.

We reduce $prec_f^{\Pi}$ to $\lfloor prec_f^{\Pi}\rfloor$ by clearing it of all universal quantification (*ie.* the $\bigwedge_j \forall\bar{a}'_j.q'_j$ conjuncts, leaving only p'). All $post_f^{\Pi}$ and $prec_g^{\Pi}$ are reduced to, respectively, $post_f^{\Pi}/E$ and $prec_g^{\Pi}/E$ by following the procedure described in 2.3. This means that all formulas handled by the framework belong to $\mathcal{L}^{\Pi}$. We also know that in each formula generated from method f, $prec_f^{\Pi}$ must hold, and therefore each conjunct $\forall\bar{a}'_j.q'_j$ (*ie.* pattern) contained in $prec_f^{\Pi}$ must hold as well.

Therefore, during its execution, our constraint generation algorithm visits the formula (rewritten for $\lfloor prec_f^{\Pi}\rfloor$, $post_f^{\Pi}/E$ and $prec_g^{\Pi}/E$) with the context described in 2.2.3. All function applications are checked for possible pattern instantiations, and each instantiation will generate a constraint that forces the model to conform to the universally quantified specifications. In other words, our algorithm handles universal quantification by reducing predicate logic formulas in $\mathcal{L}^{\Pi\mathcal{U}}$ to formulas in the propositional logic supported by $\mathcal{L}^{\Pi}$

while maintaining satisfiability and model equivalence (for a somewhat intuitive notion of equality).

To implement inductive invariance as specified in 2.2.5, we present Algorithm 2 that makes sure function-typed invocation results are correctly constrained. The assume procedure is called whenever the underlying framework generates an inductive invariant for such an invocation. Note that $f_i(\bar{x})_i$ in Algorithm 1 represents variable function applications whereas $f_i(\bar{x})_i$ represents method invocations in Algorithm 2. We use the same notation due to the high similarity between both use cases.

Algorithm 2 Inductive invariant assumption generation for method invocation unrolling

precedure assume($f_i(\bar{x})_i$)
  postcondition := $post^{\Pi}_{f_i}$
  formula := ⌊postcondition⌋
  for $\forall \bar{a}'_j . q'_j$ **in** postcondition
    constraints := $\bigwedge$ (context, $f_i(\bar{x})_i$) $\models \forall \bar{a}'_j . q'_j$
    formula := formula $\wedge$ $\rho\,(f_i(\bar{x})_i) \implies$ constraints
  done
  context := context $\cup \{f_i(\bar{x})_i\}$
  return constrain(formula)
end

2.4.1 Property preservation

We wish to establish that the satisfiability procedure of the underlying framework given in [13] remains (assuming declared function termination):

1. sound for models: every model it returns makes the formula true;
2. terminating for all formulas that are satisfiable;
3. sound for proofs: if it reports UNSAT, then there are no models;
4. terminating for all sufficiently surjective abstractions.

Since we model functions as maps, properties 2 and 4 are satisfied by previous work in [12].

Soundness for Models. Let $q \in \mathcal{L}^{\Pi\mathcal{U}}$ be a formula generated by the framework such that $q \in SAT$. Let m be a model for q, and $R(q) \in \mathcal{L}^{\Pi}$ be the result of running our algorithm on q. It is clear that m satisfies $R(q)$ as all generated constraints are guaranteed by the universal quantifiers in $prec^{\Pi}_{f}$ and all other transformations on q only *remove* constraints by clearing universal quantification. Therefore, $R(q) \in SAT$.

Now let m_R be a model for $R(q)$. It is just as clear as above that m_R *does not* satisfy q (in the general case). Indeed, only a finite number of propositional conjuncts (the constraints

generated by the algorithm) are available to constrain function variables and we therefore cannot be as general as universal quantification. However, the function variables in m_R are *sufficiently constrained* for all applications in q to be defined. This means that m_R is only incomplete in the sense that infinite input domain sizes cannot be modeled, but still suffices to prove that $q \in SAT$, and model soundness is (sufficiently) preserved. We say here that m_R is *locally* complete. This infinite model issue is in fact what kept us from using the SMT solver's native universal quantification capabilities.

Soundness for Proofs. Let $q \in \mathcal{L}^{\Pi\mathcal{U}}$ be a formula generated by the framework such that $q \in UNSAT$ and let $R(q) \in \mathcal{L}^{\Pi}$ be the result of running our algorithm on q. Assume $R(q) \in SAT$, and that model m_R satisfies $R(q)$. As $q \notin SAT$, we know that there is at least one conjunct $\forall \bar{a}'_j . q'_j$ that is not locally satisfied by m_R. In other words, for some $g \in m_R$ such that $g \subseteq q'_j$ and for some $\bar{a}$ such that the concrete application $g(\bar{a}) \subseteq q$, either 1) m_R does not define $g(\bar{a})$, or 2) $m_R \models q'_j \left[\bar{a}'_j \to \bar{a}\right]$ is false. Both cases cannot happen since 1) if $g(\bar{a}) \subseteq q$, then $g(\bar{a}) \subseteq R(q)$, as our algorithm does not remove any concrete function application, and 2) $q'_j \left[\bar{a}'_j \to \bar{a}\right]$ cannot evaluate to false for any concrete $\bar{a}$ since constraints will have been generated by the algorithm run for all possible concrete instances of q'_j. It follows that $R(q) \in UNSAT$ as well and therefore $q \in UNSAT \implies R(q) \in UNSAT$.

Now consider another pair $q \in \mathcal{L}^{\Pi\mathcal{U}}$, $R(q) \in \mathcal{L}^{\Pi}$ such that $R(q) \in UNSAT$. Assume $q \in SAT$ and we have a model m for q. Since constraints are handled by our algorithm by asserting instantiated universally quantified propositions $\forall \bar{a}'_j . q'_j$, if one holds then it is clear that $q'_j \left[\bar{a}'_j \to \bar{a}\right]$ must hold as well. It derives that if $q \in SAT$, all added constraints are also in SAT by construction.

We know from [13] that q has one of three possible shapes (for some $p \in \mathcal{L}^{\Pi}$):

1. $prec^{\Pi}_f \implies post^{\Pi}_f\,[p/\rho]$
2. $prec^{\Pi}_f \wedge p$
3. $prec^{\Pi}_f \wedge p \implies prec^{\Pi}_g$

As only $prec^{\Pi}_f$, $post^{\Pi}_f$ and $prec^{\Pi}_g$ are modified by our algorithm, only these parts of the formula need be considered.

The transformation from $prec^{\Pi}_f$ to $\lfloor prec^{\Pi}_f \rfloor$ only removes conjuncts and since $prec^{\Pi}_f$ is always a top-level conjunct or a top-level conjunct of the left-hand side in an implication, the reduction cannot break satisfiability.

The other transformation, namely p/E that is applied to $post^{\Pi}_f$ and $prec^{\Pi}_g$ *only* takes place on the right-hand side of an implication. This means that for $R(q) \in UNSAT$, the implication must stand as $true \implies false$. We have just seen that $prec^{\Pi}_f \implies \lfloor prec^{\Pi}_f \rfloor$, so the left-hand side of the implication has the same satisfiability for q and $R(q)$. As $q \in SAT$

and, given the process p/E described in 2.3, we must have a conjunct $\forall \bar{a}'_j.q'_j$ in $post^{\Pi}_f$ or $prec^{\Pi}_g$ such that $\forall \bar{a}'_j.q'_j \in SAT$ but $\exists \bar{a}.q'_j \left[\bar{a}'_j \to \bar{a}\right] \in UNSAT$, which is impossible.

Due to our special handling of $post^{\Pi}_f$ in case of functional return types, there remains a final shape introduced by our extension:

4. $prec^{\Pi}_f \implies post^{\Pi}_f [p'/\rho]$ where $p' \in \mathcal{L}^{\Pi\mathcal{U}}$

As presented in 2.2.5, our algorithm makes sure method invocations are correctly constrained by all conjuncts $\forall \bar{a}'_j.q'_j$, and we can apply the same argument as for p/E to show $R(q) \in UNSAT \implies q \in UNSAT$. Soundness for proofs is therefore preserved as well.

2.5 Integration into Leon

Higher-order functions can easily be translated to Z3 as *Z3 arrays*, a special type of map that does not require initialization, however the constraint generation algorithm presented in section 2.2 cannot simply be added to Leon as a stand-alone component. Indeed, higher-order function applications that appear during invocation unfolding must also be constrained, a task that cannot be performed ahead of time as constraints are context-dependent.

Leon manages named function invocation unfolding by transforming function definitions into a series of clauses which can then be *blocked* to control access to uninterpreted function invocations. These clauses are translated to Z3 structures and cached on creation. To unfold an invocation, the Z3 clauses assigned to its definition are updated by substituting formal arguments with the Z3 invocation arguments before being asserted by the solver. This procedure guarantees high-performance unfolding without loss of expressivity. Unfortunately, this procedure cannot be trivially extended to higher-order functions.

Constraint generation requires the fulfillment of multiple conditions to be possible. First of all, we need a precondition specifying constraints on the higher-order function applications present in the expression we wish to constrain. This can easily be provided by adding an extra argument to the solving system interface. We also need to propagate the context from one unfolding to the next in order for constraints concerning different unfoldings to be instantiated. Again, this condition is readily met by storing context as a global variable in the unfolding framework.

Regrettably, some requirements to constraint generation are not so trivially satisfied and their integration into the Leon unfolding framework is discussed in the following sections.

2.5.1 Higher-order clauses

In order for pattern matching to be possible, untranslated higher-order function applications in clauses are required during constraint generation. This issue can be solved by deferring translation until unfolding time. This unfortunately comes with a toll on performance since translation must now take place at each invocation unfolding and cannot be

cached. We can improve this naive approach by noticing that as long as the translation remains consistent, different clauses can be translated at different times.

The only part of the Leon to Z3 translation that can vary from one execution to the next is the handling of variables. The initial translation mechanism already sets up a map to guarantee consistency from one clause to the next, so we simply store this map until higher-order clause translation to guarantee overall consistency. In order to maintain high-performance for inputs that do not feature higher-order functions while simultaneously guaranteeing the availability of all higher-order constructs untranslated during constraint generation, we separate the clauses containing higher-order constructs from those that do not. Legacy clauses are handled as before while higher-order clauses are stored untranslated until unfolding time and are only translated once constraints have been generated.

Another issue pertaining to Z3 translation arises when receiving the Z3 invocation arguments. Higher-order arguments are needed untranslated in order to be replaced in higher-order clauses and used to generate constraints. Happily, the Leon solving framework already caches variable translations which lets us translate the Z3 higher-order arguments back to the necessary Leon trees, as long as the initial higher-order construct is a variable. Section 2.5.2 generalizes this approach to all higher-order constructs.

2.5.2 Anonymous functions

We have not yet addressed anonymous function handling in our system. This is because they are managed separately from the constraint generation framework. Anonymous functions are simply propagated until they are applied, whereupon the formal arguments are replaced with the application arguments in the function body.

The translation of anonymous functions to and from Z3 (necessary for Z3 arguments to named function invocation unfolding) is performed by storing a bijection between anonymous functions and fresh variables and translating the mapped variable to Z3. This scheme makes sure that the only Z3 higher-order constructs received as arguments to named function unfoldings are variables, and can therefore be translated back to Leon trees as described in section 2.5.1. To guarantee a unified and consistent approach, we (recursively) encode all non-variable higher-order arguments (*eg.* named functions) into anonymous functions.

Furthermore, a major strong-point of anonymous functions is their support for contextual closure. This means that when translating the result of anonymous function applications, we must be consistent with the translation of their defining functions. This can be partially solved by storing the relevant pieces of the Z3 variable translation map alongside the anonymous function in the bijection introduced above and adding these pieces to the map used during translation of application to Z3. Unfortunately, this solution creates a new issue: if we have multiple anonymous functions issued from the same definition but with different contexts, the pieces of translation map will conflict with each other due to matching keys. Free variables (ie. contextual variables) in newly defined anonymous functions must therefore be freshened and the translation map must be extended with these

```
public abstract class AnonymousFunction {
      public abstract Object apply(Object[] args);
}
```

Figure 2.2: Higher-order function JVM type signature

new variables. This guarantees that all anonymous functions will contain completely fresh closure variables, thus preventing any conflict while maintaining translation consistency.

2.5.3 Application evaluation

An important feature of Leon is its ability to evaluate fully determined formulas. As presented previously, we rewrite function-typed expressions to either be a function variable (argument to the current definition), or an anonymous function. We therefore observe that formulas are fully determined regarding higher-order functions as long as they do not contain any free function variables (indeed, anonymous functions are themselves completely determined).

Leon evaluator. Leon features a tree evaluator based on evaluation rules. To provide support for higher-order functions, a new evaluation rule must be introduced to evaluate higher-order function applications. As seen above, we know our function must be an anonymous function (formulas that are not fully determined are never evaluated). Evaluation is therefore trivially implemented by inlining the anonymous function body rewritten for application arguments.

One should notice that since anonymous functions enable contextual closure, whenever an anonymous function definition is encountered by the evaluation visitor, its free variables should be replaced by the evaluated contextual variables (which must be fully determined by the time we encounter an anonymous function).

JVM evaluator. To provide a more efficient implementation, a second evaluator is provided by Leon that compiles the formula to JVM byte-code before executing it and recovering the result. We introduce a new abstract class to provide a unified interface to higher-order functions (see Figure 2.2). This makes function application definition quite natural as it simply consists in the *apply* method invocation on the arguments array.

Anonymous functions are handled by defining a new class that extends the higher-order function interface for each new anonymous function encountered during compilation. To manage contextual closure, the class is provided with a constructor that takes all the contextually closed variables of the anonymous function body as arguments and viewing these as fields during the compilation of the *apply* method body. Finally, it is added to the JVM class-loader to enable instantiation at runtime. The actual anonymous function

definition is compiled as an instantiation of its associated class with the contextually closed variables taken as argument.

3 Termination

Termination proving is an unsolvable problem in the general sense, but this does not mean certain specific instances cannot be solved. Indeed, different techniques can be used to prove (or partially prove) many real-world termination problems. The number and diversity of problems which each technique can solve generally depends on its complexity and directly correlates to increased runtime. Unfortunately, it is very difficult to (efficiently) determine which technique will suffice to prove termination without actually applying it.

3.1 Modular approach

If the difference in runtimes is large (at least double), an efficient strategy is to apply each technique one after the other with increasing complexity until the problem is solved. We further improve this strategy by keeping track of partial results, obtained during previous technique applications, and injecting these into the current proof.

As we are working in a functional context, function definitions stand out as the natural building blocks to be carried from one proof to the next. Each proof *problem* consists of a set of function definitions which is refined during each technique application before being passed on. This procedure grants us clear-cut modularity with a nice system for carrying proof results over to the next technique.

Let us formalize these notions. As mentioned above, we use function definitions (noted as $FunDef$) as our basic blocks, but it is often more interesting to consider a set of interdependent definitions. Furthermore, to provide flexible capabilities to each technique, these must be able to return multiple sub-problems (used in strongly-connected termination, for example). Keeping these considerations in mind, we formalize each technique's type as

$$Set[FunDef] \implies List[Set[FunDef]],$$

where the function definition sets contained in the result are disjoint, and the union of all results is a subset of the input. Definitions that do not figure in the result have been cleared by this technique.

In this setting, *clearing* a function definition means that that particular function is not critical to program termination. This does not mean the function necessarily terminates (as it may still call – or transitively call – non-terminating functions), but we do not need to consider it explicitly when constructing later proofs and therefore does not belong to any problem.

3.1.1 Composing techniques

The framework modularity and overall performance is a consequence of the potential for composition. Simpler (and more efficient) techniques are applied initially to weed-out trivial (or easier) problems and the remaining problems are then fed to the more powerful (and more resource intensive) ones.

The refinement of a problem by a later technique application can create new opportunities for previous ones. For example, if a powerful technique eliminates one function definition in a *strongly connected component* (*ie.* clears the function), this might split the remaining set into multiple distinct sub-components that do not interact with each other and should therefore be treated as distinct problems. To avoid missing these opportunities, each successful problem refinement sends the results back to the beginning of the queue.

The system thus described is an adaptation of the *Dependency Pair Framework* [5] for TRS to a functional programming setting and gives rise to Algorithm 3. If a termination problem has not been solved after having run through the whole pipeline, it is added to an unsolvable set and ignored during the rest of the algorithm execution. Indeed, the modularity of the system ensures that problems have no cross-dependencies.

By definition, problem size is strictly under-bounded by zero and refined sub-problems are subsets of source problem. Since refinements can only take place finitely many times on each problem, we can see that Algorithm 3 is guaranteed terminating.

Algorithm 3 Pseudo-code for the adaptation of the *Dependency Pair Framework* for TRS to a functional setting.

```
processors := initial processors
problems := initial functions
unsolvable := ∅
while (problems ≠ ∅) {
  (problem, index) = problems.head
  result := processors[index](problem)
  if (result = {problem}) {
    if (index = processors.size - 1)
      unsolvable += {problem}
    else
      problems += {(problem, index + 1)}
  } else {
    problems += {(p, 0) | p ∈ result}
  }
}
```

3.1.2 Using verification

As we are building on a fully fledged verification framework (Leon), it would be quite interesting to be able to use its capabilities during termination proving. This enables us to reason about formulas, thus providing a welcome abstraction from syntax trees.

One of the main strengths of Leon is its ability to handle function invocations through unfolding as described in [13], and to assume *postconditions* (an invariant on function results). Sadly, the inductive nature of these *postconditions* requires the declaring function to terminate in order for them to be guaranteed. Therefore, these *must* be ignored during the unfolding of functions that have not (yet) been proved terminating.

Our definition for cleared functions specifically noted that these are not guaranteed to terminate, but simply are not critical to solving the current problem. To determine termination, we must therefore check that the function in question is cleared AND that it does not transitively call any function that still lies in the problems queue or in the unsolvable set. This means that problems are now interdependent and implies a certain loss in framework modularity. In practice, the only modification required in the solving algorithm is that when new refinements successfully take place, we must reenter the previously unsolvable definitions into the problems queue and clear the set, since the processors now have more information, and may be able to prove termination this time around (see Algorithm 4). We also introduce the refined problems at the beginning of the queue since the more the overall proof is advanced, the more the power of the verification framework can be harnessed. The algorithm is still guaranteed to converge as reset can only happen finitely many times (it requires clearing a definition).

Algorithm 4 Modular solving algorithm presented in Algorithm 3 when using the underlying verification framework.

```
processors := initial processors
problems := initial functions
unsolved := ∅
while (problems ≠ ∅) {
  (problem, index) = problems.head
  result := processors[index](problem)
  if (result = {problem}) {
    if (index = processors.size - 1)
      unsolved += {problem}
    else
      problems += {(problem, index + 1)}
  } else {
    problems := {(p, 0) | p ∈ result} + problems + {(unsolved, 0)}
    unsolved := ∅
  }
}
```

Aside from the necessary loss of modularity, unwanted function unfolding poses no serious problem. Indeed, it suffices to remove the *postconditions* specified on these functions. Since the underlying verification system is counter-example based, this can only weaken the termination condition, which is why we may improve solutions by sending previously unsolvable problems back through the pipeline. Once a function has been proved terminating, its inductive invariant may be reinstated to bolster proof construction. However, this implies that the validity of our termination proof depends on the validity of the invariant. As we have shown the function is terminating, we can simply use the underlying framework to prove the invariant holds (as it is complete given termination).

It should also be noted that the heuristics used to determine problem solving order can have a substantial impact on the number of times unsolvable problems will be sent back into the pipeline, but will not impact the power of the solving framework. These heuristics are kept simple here (breadth-first search in Algorithms 3 and 4) and are not further discussed in this paper.

3.2 Termination techniques

The modular framework we introduced above only makes sense when composing multiple techniques. Five *processors* (read techniques) were therefore implemented and hierarchically integrated into our system:

Component processor relies on the program call graph in order to group function definitions into strongly connected components and filters out definitions that are transitively non-calling.

Recursion processor searches for structural decreasing of same argument in strictly self-recursive functions.

Relation processor compares the structural size of definition inputs with that of sub-call inputs, searching for strict decreasing or transitively strict decreasing.

Chain processor discovers call chains with start- and end-points in a given definition and searches for convergence towards terminating CFG nodes through structural size reduction, numeric progression or boolean inversion.

Loop processor searches for loops in call chains (again with start- and end-points in same definition) with identical arguments at start and end of the chain that imply non-termination.

The first two processors were taken as such from previous work, and will not be discussed in this paper. They serve to illustrate how easily extra processors can be integrated into this framework.

3.2.1 Structural size

We must now introduce the notion of *structural size*, which is the key to understanding the **relation processor** and (at least part of) the **chain processor**. We recursively define structural size on types as

$$size(x) = \begin{cases} \sum_{i=1}^{N} size(x_i) & if\, x : Tuple_N \\ 1 + \sum_{i=1}^{N} size(x.field_i) & if\, x : ADT_N \\ 0 & otherwise \end{cases}$$

This function is guaranteed to terminate because the only recursively definable types, namely algebraic data types (ADTs), have a loop-free structure by design. Furthermore, the function is monotonic with respect to ADT inclusion ($x.field_i \subset x$), which is a well founded order (can be easily proven by induction). Moreover, the function is lower-bounded by 0, a much desired quality in our case. We can now define a total order on data structures as $size(x_1) < size(x_2)$. Since the structural size function is lower-bounded by 0, it is impossible to generate infinite chains with decreasing size! This property is the main driving force behind *size-change termination* proofs [10].

As the *size* function described above is a synthetic program construct (*ie.* generated by the framework), it cannot be referenced by user-defined functions in the program. This unfortunately makes size reduction proofs much harder as no invariants on *size* are available to bolster proving capabilities. To alleviate this issue, we can automatically generate different types of invariants that will help build such proofs.

Inductive invariants on *size*. We know by definition that

$$\forall x.size(x) \geq 0 \text{ and } \forall x : ADT.size(x) > 0.$$

These observations can be used to provide an inductive invariant on the *size* function. We therefore extend $post^{\Pi}_{size}$ with the conjunct $size(\bar{a}_{size}) >_{:ADT} 0$ where $>_{:ADT}$ represents strict or soft inequality depending on whether the argument to *size* is an ADT or not.

We can further strengthen the invariant by adding bound constraints on recursive sub-calls to *size* in $impl^{\Pi}_{size}$. These are provided by

$$\bigwedge_{i=0}^{N} size(\bar{x})_i >_{:ADT} 0 \text{ where } \forall i \in [0, N].size(\bar{x})_i \subseteq impl^{\Pi}_{size}.$$

The truth of these new assertions is inductively guaranteed, and they ensure that the values taken by uninterpreted sub-calls in $impl^{\Pi}_{size}$ make (more) sense as inductive invariant assumption only takes place *after* definition unfolding in the underlying framework.

Inductive invariants on f. It can also be interesting for function $f \in \Pi$ to provide inductive invariants that constrain the relation between the *size* of inputs and outputs of f (*eg.* $size(\bar{a}_f) < size(f(\bar{a}_f)))$ as this will provide valuable information for proofs where sub-call arguments are function results. Such instances are practically impossible to prove terminating without these *size* invariants.

Since we are working atop a verification framework, we can use it when providing these invariants by attempting verification of a potential inductive invariant that states a relation between inputs and outputs of function f. If the verification succeeds, we can maintain the invariant, otherwise it must be dropped. As verification is only complete given termination, we can only provide these invariants on functions that have already been proved terminating.

From [13], we know that verification depends on the proof of three different properties:

1. $prec_f^\Pi \implies post_f^\Pi \left[impl_f^\Pi/\rho\right]$;
2. for any call $f_i(\bar{x})_i \subseteq impl_f^\Pi$, the formula $\rho(f_i(\bar{x})_i) \implies prec_{f_i}^\Pi$ must hold;
3. for each pattern-matching expression, the patterns must be shown to cover all possible inputs under the path condition.

Properties 2 and 3 are not modified by our extension of the inductive invariant (*ie.* $post_f^\Pi$), so we need not consider these. Therefore, to verify whether our new inductive invariant holds or not, it suffices to prove property 1 where $post_f^\Pi$ is replaced by $post_f^\Pi \wedge size(\bar{a}_f) < size(f(\bar{a}_f))$. Note that soft decreasing is also a worthwhile property to consider if strict decreasing proof fails.

3.2.2 Relation processor

Structural size provides us with a comparison metric between *any* two values, not only those which share a type. This means we can compare the arguments of any sub-call with those of the caller. If the *tuple* of sub-call arguments has strictly smaller structural size than the *tuple* of caller arguments for all sub-calls in a definition, then we can mark it as non-critical for program termination. Indeed, any program loop leading to non-termination which contains a definition with decreasing argument structural size for all sub-calls must also contain (at least) one definition for which this property does not hold (otherwise, we have a lower-bounded infinitely decreasing chain: contradiction).

We can formalize the non-criticality of a function f as follows. First of all, let the set $\{f_i(\bar{x})_i \,|\, 0 \leq i \leq N \wedge f_i \in \Pi\}$ consist of all method invocations in $impl_f^\Pi$ (note that f_i may equal f_j for $i \neq j$), and $\alpha(f_i(\bar{x})_i)$ be the tuple of argument expressions to invocation $f_i(\bar{x})_i$. We obtain the following general formula for non-criticality of function f:

$$\bigwedge_{i=0}^{N} (\rho(f_i(\bar{x})_i) \implies size(\alpha(f_i(\bar{x})_i)) < size(\bar{a}_f))$$

We can even slightly relax the structural size decreasing constraint by requiring *transitive decreasing* instead of strict decreasing. If the structural size of caller arguments is less than or equal to that of a sub-call (*ie.* soft decreasing) and is strictly decreasing in sub-call definition, we say that the argument structural size is *transitively decreasing*. This concept can be extended over arbitrarily large invocation trees as long as the leaf nodes feature strict decreasing. We compute this tree by fixed point iteration, starting from the set of strictly decreasing definitions and adding all definitions that *only* call functions present in the decreasing set (and feature soft decreasing for all invocations in body). Any loop containing a definition with transitively decreasing structural size of arguments must contain (at least) one definition featuring strict decreasing (all call chains starting at a definition with transitively decreasing argument structural size must pass through a tree leaf or be non-looping because of loop-free tree structure). We can therefore apply the same argument as previously to show that such definitions are non-critical.

3.2.3 Chain processor

Non-termination arises when a program gets stuck in a never-ending loop. Therefore, an excellent way of proving termination is showing that no loop in the program can generate an infinite sequence. Such a loop *must* contain a critical function (else we have termination, as exposed previously). These observations are the basis upon which the **chain processor** is built.

Chains and clusters. Let us start by defining the notion of *chain* that will be used throughout this section. A chain c is a sequence $f_0(\bar{x})_0, \ldots, f_N(\bar{x})_N$ of function invocations such that $f_0(\bar{x})_0 \subseteq impl^{\Pi}_{f_N}$ and $f_i(\bar{x})_i \subseteq impl^{\Pi}_{f_{i-1}}$ for $0 < i \leq N$. Note that $f_i = f_j$ is possible for $i \neq j$; we use this notation to easily refer to functions by index. It is clear that a homomorphism exists between loops and chain equivalence classes, so each loop can be represented by *at least* one chain.

We define $\rho\,(c)$ as the SAT formula encoding of a chain. We bind the control-flow-graph encoding $\rho\,(f_i(\bar{x})_i)$ of each invocation $f_i(\bar{x})_i$ with the next invocation $f_{i+1}(\bar{x})_{i+1}$ by requiring equality between the free variables $\bar{a}_{f_{i+1}}$ in f_{i+1} (*ie.* formal arguments) and the invocation arguments $\alpha\,(f_i(\bar{x})_i)$. This gives us the formulas $\rho\,(f_i(\bar{x})_i) \wedge \left(\bar{a}_{f_{i+1}} = \alpha\,(f_i(\bar{x})_i)\right)$ for each invocation f_i where $0 \leq i < N$. As f_i may equal f_j for $i \neq j$, the free variables in each formula must be freshened, but we must still maintain bindings between invocation $f_i(\bar{x})_i$ and $f_{i+1}(\bar{x})_{i+1}$. We achieve this by building a map F from variables $\bar{a}_{f_i}$ to their fresh counterparts, thus obtaining the formula

$$\rho_T(c) = \bigwedge_{i=0}^{N} F\left(\rho\,(f_i(\bar{x})_i) \wedge \left(\bar{a}_{f_{i+1}} = \alpha\,(f_i(\bar{x})_i)\right)\right).$$

We must also introduce a final constraint that ensures the formula $\rho\,(f_0(\bar{x})_0)$ is still in SAT

after a run through the loop, which gives us

$$\begin{aligned} F'(p) &= p\left[\bar{a}_{f_0} \rightarrow F\left(\alpha\left(f_N(x)_N\right)\right)\right] \\ \rho_L(c) &= F'\left(\rho\left(f_0(\bar{x})_0\right) \wedge \left(a_{f_0} = \alpha\left(f_0(\bar{x})_0\right)\right)\right). \end{aligned}$$

The complete formula is clearly obtained as the conjunction $\rho(c) = \rho_T(c) \wedge \rho_L(c)$.

For any two chains a, b that share a same f_0, we define chain composition $a \cdot b$ as the chain resulting from the execution of a and then b. Formally,

$$a \cdot b = f_{0_a}(\bar{x})_{0_a}, \ldots, f_{N_a}(\bar{x})_{N_a}, f_{0_b}(\bar{x})_{0_b}, \ldots, f_{N_b}(\bar{x})_{N_b}.$$

Let C_f be the set of all chains starting in a given function f such that for any index pairs $i, j \in [0, N]$, we have $f_i(\bar{x})_i = f_j(\bar{x})_j \implies i = j$ (we consider full invocation equality and not simply result equality here). Notice that there is a finite number of chains in C_f and that C_f *does not* contain all possible loops starting in f (consider the loop going twice through chain $a \in C_f$, then once through chain $b \in C_f$ before starting over again). Let us also define L_f as the set of chains which start in f and that do not contain sub-loops starting in function $g \neq f$. We see that $\forall c \in L_f, \exists c_0, \ldots, c_n \in C_f$ such that $c = c_0 \cdot \ldots \cdot c_n$.

We can now define the notion of *chain re-entrance*. Let $R_f \subseteq C_f \times C_f$ be the re-entrance relation such that for any $a, b \in C_f$, $(a, b) \in R$ if and only if $\rho(a) \wedge \rho\left(f_{0_b}(\bar{x})_{0_b}\right) \in SAT$. Note that for any $a, b \in C_f$, $\rho(a \cdot b) \in SAT$ only if $(a, b) \in R$. Indeed, if $(a, b) \notin R$, then there are no possible inputs to a such that it can re-enter b, so the program can never execute a and then b.

Given the re-entrance relation R_f, for $c \in C_f$ we define the *cluster* $\gamma(c)$ as the least fixed point of R_f on c. An interesting property of $\gamma(c)$ is that $\forall c_L \in L_f$ such that c_L starts with c, $\exists c_1, \ldots, c_n \in \gamma(c)$ such that $c_L = c \cdot c_1 \cdot \ldots \cdot c_n$. Note that $\forall c_L \in L_f, \exists c \in C_f$ such that c_L starts with c. Furthermore, given $c_n \in \gamma(c)$ such that c_L ends with c_n,

$$\begin{gathered} \{a \in \gamma(c) \mid \rho\left(c_L \cdot a\right) \in SAT\} \\ \subseteq \\ \{a \in \gamma(c) \mid (c_n, a) \in R\} \end{gathered}$$

follows from $\rho(c_L) \implies \rho\left(c_L[0 : n-1]\right) \wedge \rho\left(c_n\right)$ by construction. Moreover, ρ only adds more constraints, so $\rho\left(c_n\right) \wedge \rho\left(f_{0_a}(\bar{x})_{0_a}\right) \in SAT$ if $\rho\left(c_n \cdot a\right) \in SAT$, which in turn is implied by $\rho\left(c_L \cdot a\right) \in SAT$. This means that if we are able to build a termination proof based on satisfiability of a formula concerning all $a \in \gamma(c)$ for all $c \in C_f$, then c_L is not critical to termination for any $c_L \in L_f$.

Proving cluster termination. In order to prove that a certain chain is terminating, we must show that it is impossible to build an infinite sequence of invocations from that chain. We can show this by considering structural size reduction, as in the **relation processor**.

But in this case, we can greatly improve the finesse of the comparison by using the fact that the chain loops back to its first component, so we can compare input arguments before and after a run through the chain. Therefore, instead of proving size decrease for the argument *tuple*, it suffices to show that one single argument decreases.

This condition can be further relaxed since *any* decreasing sequence lower-bounded by 0 is sufficient to prove termination. We can therefore consider *tuple* fields individually, for example. This observation can be generalized making a distinction between two categories of types: *container types* and *recursive types*. Container types encompass *tuples* and ADTs that have no siblings and are not recursively defined. Recursive types are all other ADTs. Since container types basically behave like *tuples*, we can show termination by considering single fields of the container type (this statement can be inductively expanded since container types are non-recursive).

In fact, it is not necessary to limit ourselves to structural decrease as the convergence criteria. Indeed, since we now have a context (namely the start and end definition of the chain), we can also discover numeric convergence criteria. These can be extracted from the definition body by considering CFG nodes where splits take place and checking for *Integer* bound conditions selecting a branch that terminates. We can even inline chain invocation bodies to find more end-points.

We now have a tool to prove single chain termination, but what we are interested in is cluster termination. The previous technique can be easily extended to whole clusters by requiring the same sub-component of the argument *tuple* to converge for all chains in the cluster. This means that for each cluster $\gamma(c)$, each $c_c \in \gamma(c)$ will have the *same* decreasing sub-component and therefore $c_{c,1} \cdot \ldots \cdot c_{c,n}$ converges for any $c_{c,0}, \ldots, c_{c,n} \in \gamma(c)$.

Clearing definitions. It is clear by cluster construction that $\forall a, b \in C_f$, we either have $\gamma(a) = \gamma(b)$ or $\gamma(a) \cap \gamma(b) = \emptyset$. Furthermore, if $C_f = \{c_0, \ldots, c_N\}$, then $\forall a \in C_f$ such that $a \notin \bigcup_{i=0}^{N} \gamma(c_i)$, a cannot be part of a possible loop starting in f since $\forall b \in C_f$, $\rho(a) \wedge \rho(f_{0_b}(\bar{x})_{0_b}) \in UNSAT$, and all other possible loops are of the shape $c_0 \cdot \ldots \cdot c_n$ for $c_0, \ldots, c_n \in \gamma(c_0)$. Therefore, if we show termination of $\gamma(c)$ for all $c \in C_f$, we have termination for L_f.

We can determine that a function f is cleared when non-termination can only be implied by non-termination in another function $g \neq f$. We state that if $\forall c_L \in L_f$, c_L is non-critical, then the function definition f is non-critical. The reasoning behind this affirmation is as follows:

We start by considering all chains starting in f that contain no sub-loops starting in $g \neq f$. All of these chains belong to L_f by definition and are non-critical by our starting assumption. Remaining chains *must* contain a sub-loop that starts and ends in another definition $g \neq f$ (*ie.* for any remaining chain c, $\exists c_g \in L_g$ such that $c_g \subseteq c$). If that chain is non-terminating, then non-termination of f is implied by criticality of g.

If the chain c_g is terminating, then we have a limit case when the convergence criteria are different and each loop works against the other. But we are guaranteed that if the work

is equal (increments and decrements are symmetrical), then there is a chain originating in each definition that is non-terminating (since it goes through both loops once), illustrated in Figure 3.1a. If the work is asymmetrical, then either one chain or the other belongs to a critical cluster. Indeed, one cannot prove convergence for the same criteria in sub-loop and containing loop, as seen in Figure 3.1b.

3.2.4 Loop processor

All we have done up until now is introduce techniques to prove termination. But it can also be useful to consider the converse case, namely proving non-termination, especially if we can present precise information about the issue like an example set of arguments causing infinite execution.

This can be easily implemented with our current framework. Indeed, we already have a nice formalism to reason about a large class of loops (namely chains). If for any chain $c \in C_f$, $\rho(c) \land F'(\bar{a}_{f_0} = \alpha(f_0(\bar{x})_0)) \in SAT$ holds (*cf.* 3.2.3 for the definition of $F'(p)$), then we know that there exists an infinite execution through c. Since the underlying model provides us with a model when proving SAT, we also obtain a set of input arguments to the chain that create an infinite execution.

We can further improve this technique by considering chain compositions $c_0 \cdot \ldots \cdot c_n$ (for finite $c_0, \ldots, c_n \in C_f$). However, since this processor is not guaranteed to prove non-termination even when it may exist (*eg.* does not detect count-to-infinity) and there is an infinite number of different compositions, in practice, we must limit ourselves to a certain finite subset lest the processor run forever.

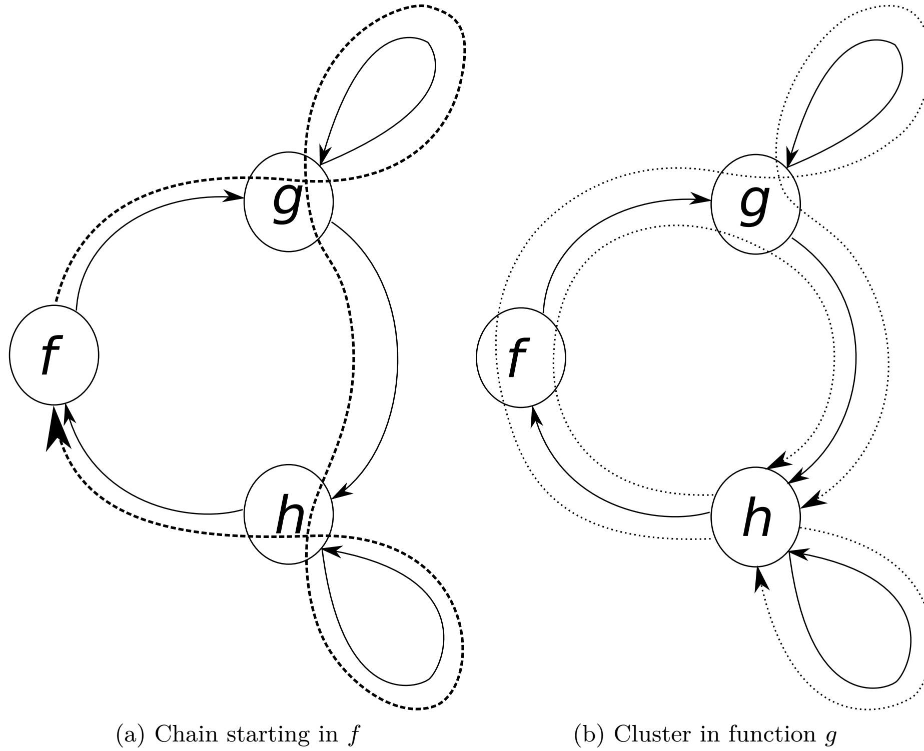
3.3 Handling higher-order functions

We have not mentioned up to now how higher-order functions can be managed during termination proving. As observed previously, non-termination can only arise when a program execution gets stuck in a never-ending loop, and the only constructs that can generate loops in a functional context are function invocations.

3.3.1 Proving termination

When extending a first-order functional context with higher-order functions, we add a new type of function: variable functions. The concrete instances of variable functions are *always* anonymous functions (modulo rewriting function-typed method invocations or methods as anonymous functions, which preserves semantics in our functional call-by-value context). Anonymous functions that do not contain function applications – variable or method – are guaranteed terminating as they cannot generate loops.

When anonymous functions contain method invocations, this means these invocations will *only* be evaluated when the anonymous function is applied, generally with arguments depending on those passed to the anonymous function. We can therefore prove termination



(a) Chain starting in f (b) Cluster in function g

Figure 3.1: Illustration of clearing limit-cases handled by complementarity chains and clusters. Let us assume that the loops in g and h compensate each other. If they do so by looping an equal number of times, then proving that the loop depicted in 3.1a terminates implies termination for any number of loops through g and h as long as they cancel each other out. If the loop in h needs to loop more times than the other, then the cluster in 3.1b will not be proved terminating (as the small loop in h will not make the same arguments decrease as the large one that loops through g). By symmetry, we have the same property if g has to loop more to compensate for h.

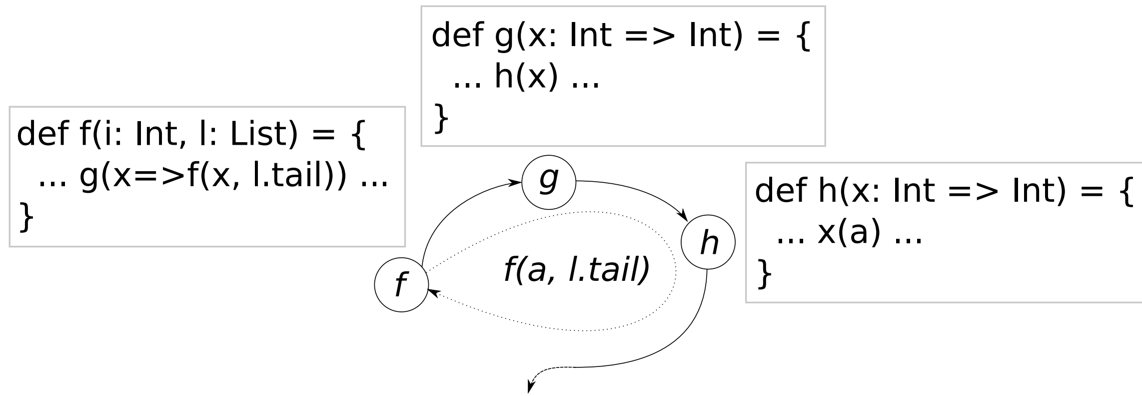


Figure 3.2: Example of higher-order termination. Termination of $f(x,\, l.tail)$ implies termination of $f(a,\, l.tail)$ regardless of the path taken to $f(a,\, l.tail)$ since x is a generalization of a.

by building the complete function application graph and verifying each set of arguments to make sure they imply termination. However, loops generated by method invocations in anonymous functions *only* depend on the path taken to function application in the constraints it will add to the application arguments. Therefore, termination for method invocation regardless of application arguments implies termination for all concrete applications (as in 2.3). Such a proof is harder to build as we must do so with fewer constraints, but it has a valuable feature: anonymous functions that have been cleared remain so even in the case of program extension. This means that all termination problems that have been solved for our program Π need not be considered in program Π' for all $f' \in \Pi'$ such that $f' \in \Pi$.

This approach also makes for a much simpler implementation. Indeed, we can completely ignore variable functions (as they are guaranteed terminating at creation) and presence or not of considered method invocations in anonymous function definitions. We disregard concrete application arguments by treating formal arguments as free variables.

3.3.2 Proving non-termination

Unlike termination proving, concrete arguments to variable function applications do impact non-termination in an unavoidable way. Indeed, the free variables introduced by anonymous functions make loops formulas *easier* to satisfy by the underlying framework, and thus often depend on future concrete applications that may never take place in the current program.

As we (optimistically) focused on termination proving in this work, we left the question of non-termination in the context of higher-order functions open and simply noted loops going through anonymous functions as unreliable. Non-termination in a higher-order

functional context would be an interesting extension to this work.

3.3.3 Providing invariants

Although we disregard application sites for anonymous functions to simplify proofs and make them more general, we can still build invariants on their argument sizes in certain cases. If we can provide a relation on *all* applications of a certain function-typed argument in the body of a definition, then we know this invariant will hold on arguments of the higher-order functions passed to invocations of this definition. Note that these relations can be built transitively.

We use this observation by building invariants regarding the *size* of definition arguments and higher-order function arguments which can then be inlined into the path conditions considered for termination proving. This greatly increases proof capabilities in the context of higher-order functions as these *size* relations appear in many – if not most – catamorphisms and are therefore ubiquitous in higher-order functional programming.

4 Generic Type Polymorphism

We extend the syntax with type parameters, largely by following the Scala language specifications (see Figure 4.1). As the underlying type system provides only a very simple inheritance scheme, type constraints are not supported in our implementation. Furthermore, to ensure backwards compatibility to different pre-existing features, both parent and child algebraic data types must be completely defined by their children, respectively parent. In other words, a bijective mapping between parent and child type parameters *must* exist for the ADT definition to be valid. The simplest way to ensure this is to only accept ADTs where both parents and children have an equal number of (constrained and simple) type parameters.

4.1 Verification

The SMT solver that powers most of the framework proofs does not support genericity. However, this feature can be simulated. Indeed, the only generic control-flow construct featured by the framework is named function invocations. We can therefore define the typed function $f_{[:T]}$ where the type parameters T_f of f are mapped to concrete type parameters T issued from an invocation $f_{[:T]}(\bar{x})$. For any typed function $f_{[:T]}$, we have $impl^{\Pi}_{f_{[:T]}}$, $prec^{\Pi}_{f_{[:T]}}$ and $post^{\Pi}_{f_{[:T]}}$ issued respectively from $impl^{\Pi}_{f}$, $prec^{\Pi}_{f}$ and $post^{\Pi}_{f}$ by applying the type mapping $[T_f \rightarrow T]$ to the formula. Furthermore, for any formula q and typed function

$$
\begin{array}{rcl}
definition & ::= & \textbf{abstract class}\ id\ tparams \\
 & & \mid \textbf{case class}\ id\ tparams\ \langle\ \textbf{extends}\ id\ tparams\ \rangle? \\
 & & \mid fundef \\
fundef & ::= & \textbf{def}\ id\ tparams\ (\ decls\ )\ :\ type = \{ \\
 & & \quad \langle\ \textbf{require}\ (\ expr\ )\ \rangle? \\
 & & \quad expr \\
 & & \}\ \langle\ \textbf{ensuring}\ (\ id\ \Rightarrow\ expr\ )\ \rangle? \\
tparams & ::= & \epsilon \mid [\ id\ \langle\ ,\ id\ \rangle^*\ ] \\
expr & ::= & PureScala\ expr \\
 & & \mid id\ [\ type\ \langle\ ,\ type\ \rangle^*\ ]\ (\ \langle\ expr\ \langle\ ,\ expr\ \rangle^*\ \rangle?) \\
type & ::= & PureScala\ type \\
 & & \mid id\ [\ type\ \langle\ ,\ type\ \rangle^*\ ]
\end{array}
$$

Figure 4.1: Syntax extensions for generic type polymorphism support

$f_{[:T]}$, we can also define

$$
\begin{array}{rcl}
\tau(q) & = & \{T \mid x : T \land x \subseteq q\} \\
\tau\left(f_{[:T]}\right) & = & \tau\left(impl^{\Pi}_{f_{[:T]}}\right) \cup \tau\left(prec^{\Pi}_{f_{[:T]}}\right) \cup \tau\left(post^{\Pi}_{f_{[:T]}}\right) \\
\Gamma(q) & = & \left\{f_{i[:T_i]} \mid f_{i[:T_i]}(\bar{x})_i \subseteq q\right\} \\
\Gamma\left(f_{[:T]}\right) & = & \Gamma\left(impl^{\Pi}_{f_{[:T]}}\right) \cup \Gamma\left(prec^{\Pi}_{f_{[:T]}}\right) \cup \Gamma\left(post^{\Pi}_{f_{[:T]}}\right).
\end{array}
$$

For any formula q that will be fed to the SMT solver, we define the transitive type set of q as the least fix-point of Γ (trivially extended to sets by union) over $\Gamma(q)$ and finally running τ over the resulting set of typed functions. Once we have the transitive type set, generic types can be mapped to equivalent non-generic types which are supported by the framework. However, we cannot build the fix-point directly as it is not guaranteed to terminate. What we can build is a limited fix-point that stops after a certain number of iterations. As the underlying framework progressively unrolls the function definitions (like the fix-point iteration), we can optimistically compute the fix-point to a certain limit and if this limit is passed without a proof having been constructed, we extend the fix-point limit and build a new generic to non-generic mapping before resuming proof construction.

4.2 Evaluation

As introduced in 2.5.3, the underlying framework provides evaluation features as well as verification. Since values that are generically typed have no operators defined (except for equality), no rule need be added to the tree walking evaluator. We simply need to make sure type parameter mappings are correctly propagated during the visit. We therefore modify the named function invocation rule (as only named function invocations propagate types) by visiting $impl^{\Pi}_{f_{[:T]}}$, $prec^{\Pi}_{f_{[:T]}}$ and $post^{\Pi}_{f_{[:T]}}$ instead of their respective counterparts during invocation unfolding.

JVM evaluation is handled by generalizing type parameters to the $Object$ JVM type, thus performing type erasure. However, as types are unfortunately no longer fully determined at compile time, we must now perform boxing and unboxing on primitive typed values when these are passed off as $Objects$. Furthermore, we now also require an *equals* method defined on *all* types as any value can be injected into an object that will require checking equality (and Leon features true equality, unlike the JVM). This can be implemented through wrapper classes for native constructs (*eg.* arrays) that will provide the necessary method at the unfortunate cost of performance. We chose to focus on soundness here and not performance, so instead of implementing a complex type analysis scheme to encapsulate and decapsulate arrays only when needed, we simply dropped native arrays and replaced them with a wrapper class that provides sound array operations regardless of concrete type.

5 Conclusion

We have presented extensions to a functional verification framework that add support for higher-order functions, complex termination analysis and generic type polymorphism. Each feature was furthermore implemented and evaluated on diverse functional programs.

Higher-order verification was extremely conclusive at detecting erroneous specifications, but was not so good at proofs. Indeed, the lack of meaningful inductive invariants that one can specify on higher-order functions unfortunately curbs the power of the verification framework. However, when the induction scheme is manually specified, success rate increases drastically. This approach enabled us to prove relations between many data-structure operators such as map, $filter$, $exists$, $forall$, etc. It should also be noted that pattern based universal quantification as introduced by the **forall** construct provides a nice foundation for introducing lemmas into Leon in future work.

Termination analysis proved surprisingly effective, with all pre-existing PureScala ADT based code samples proved terminating by the extension. The size invariant builder for higher-order function applications permitted proofs of diverse tree-walking and transforming functions based on argument partial functions. A few rather synthetic examples could not be proven terminating, but, to offer a certain effectiveness metric, the native ACL2 [8] termination prover also failed on these. It would be interesting to complement the

few automated invariant identification schemes we presented with more powerful logic to fill in the gaps still missing in some proofs. More techniques could also be added to the pipeline, such as quorum chain termination proving or a summary based relation processor as featured by T2 [3].

Finally, generic data-structure verification support increases the input space dramatically while coming at a decidedly low cost, particularly since the transitive type set is actually finite and quite shallow in most real-world programs.

References

[1] Andreas Abel. foetus–termination checker for simple functional programs. *Programming Lab Report*, 1998.

[2] Régis Blanc, Viktor Kuncak, Etienne Kneuss, and Philippe Suter. An overview of the leon verification system: Verification by translation to recursive functions. In *Proceedings of the 4th Workshop on Scala*, page 1. ACM, 2013.

[3] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Summarization for termination: no return! *Formal Methods in System Design*, 35(3):369–387, 2009.

[4] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[5] Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 301–331. Springer, 2005.

[6] Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N Freund, and Cormac Flanagan. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop*, pages 93–104, 2006.

[7] John Hughes. *Functional Programming Languages and Computer Architecture: 5th ACM Conference. Cambridge, MA, USA, August 26-30, 1991 Proceedings*, volume 523. Springer, 1991.

[8] Matt Kaufmann and J Strother Moore. Acl2 homepage. *See URL http://www. cs. utexas. edu/users/moore/acl2*, 2006.

[9] Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. Scala to the power of z3: Integrating smt and programming. In *Automated Deduction–CADE-23*, pages 400–406. Springer, 2011.

[10] Chin Soon Lee, Neil D Jones, and Amir M Ben-Amram. The size-change principle for program termination. In *ACM SIGPLAN Notices*, volume 36, pages 81–92. ACM, 2001.

[11] Martin Nordio, Cristiano Calcagno, Bertrand Meyer, Peter Müller, and Julian Tschannen. Reasoning about function objects. In *Objects, Models, Components, Patterns*, pages 79–96. Springer, 2010.

[12] Philippe Suter, Mirco Dotta, and Viktor Kuncak. Decision procedures for algebraic data types with abstractions. In *Acm Sigplan Notices*, volume 45, pages 199–210. ACM, 2010.

[13] Philippe Suter, Ali Sinan Köksal, and Viktor Kuncak. Satisfiability modulo recursive programs. In *Static Analysis*, pages 298–315. Springer, 2011.

[14] Niki Vazou, Patrick M Rondon, and Ranjit Jhala. Abstract refinement types. In *Programming Languages and Systems*, pages 209–228. Springer, 2013.