A Sound Strategy to Compile General Recursion into Finite Depth Pattern Matching

Maycon J. J. Amaro[1][0000−0001−7684−8428], Samuel S. Feitosa[2][0000−0002−9485−4845], and Rodrigo G. Ribeiro[1][0000−0003−0131−5154]

[1] Universidade Federal de Ouro Preto, Ouro Preto — MG, Brazil
[2] Universidade Federal da Fronteira Sul, Chapecó — SC, Brazil

Abstract. Programming languages are popular and diverse, and the convenience of programmatically changing the behavior of complex systems is attractive even for the ones with stringent security requirements, which often impose restrictions on the acceptable programs. A very common restriction is that the program must terminate, which is very hard to check because the Halting Problem is undecidable. In this work, we proposed a technique to unroll recursive programs in functional languages to create terminating versions of them. We prove that our strategy itself is guaranteed to terminate. We also formalize term generation and run property-based tests to build confidence that the semantics is preserved through the transformation. Our strategy can be used to compile general purpose functional languages to restrictive targets such as the eBPF and smart contracts for blockchain networks.

Keywords: program transformation · recursion · program generation

1 Introduction

Looping statements and recursion are one of the most common features of programming languages, but they also require a lot of caution. Non-termination is at best an annoying situation, and at worst a serious security or logical concern. Some compilers and technologies put a lot effort in guaranteeing that no program will run forever, inevitably being very restricted due to the undecidability of the Halting Problem [22]. Some examples include dependently typed languages that are used as proof assistants, such as Coq [13] and Agda [18]; smart contract languages for blockchain systems [14] and the technology to run sandboxed programs in the Linux Kernel—the eBPF (extended Berkeley Packet Filter).

eBPF [7] is an interesting case to explore, because its verifier will reject any program that has a back jump. In other words, it will reject any form of repetition[3], be it iterative or recursive. Their motivation in doing so is understandable: running programs in the Operating System's kernel requires a lot of caution. One

[3] Version 5.3 and higher has support for bounded loops only

mistake could bring the system down. Allowing potential non-terminating programs is a breach that ill-intentioned users could explore to perform Denial of Service attacks.

While imperative languages have additional constructs for repetition, functional languages can only count with recursion for repeating computations. Bounded loops are one easy way to walk around the restriction that all programs must terminate, but using functional languages to write programs targeting eBPF requires better strategies. Notice that ensuring termination alone is not sufficient. A program with repetition, even with a proof that it terminates, would still be rejected by eBPF. Termination must be syntactically assured.

In this work, we present an algorithm to transform recursive functions into finite depth pattern matching functions with an *equivalent* semantics, in the sense that both functions must yield the same results when the recursive function halts and the non-recursive function, with the given input, has enough nested pattern matching constructions to produce a value. We also present a sound strategy to generate random terminating programs so our algorithm can be properly tested. More specifically, we make the following contributions:

- We present System R, a core language with recursion and its unrolling algorithm.
- We present System L, a core language with no recursion and show how to compile system R programs into equivalent System L programs.
- We describe a sound algorithm to generate random well-typed terminating programs for the System R language.
- We formally demonstrate some properties regarding the presented algorithms. Property-based tests are applied otherwise.

2 Basic Definitions

We consider the simply typed λ-calculus, frequently represented as $\lambda_{\rightarrow}$ [20], extended with natural numbers, pattern matching over naturals and recursion, as defined in [23]. Its syntax is described by the following context-free grammar:

$$\begin{aligned}
\tau &::= \mathbb{N} \mid \tau \rightarrow \tau \\
e &::= \text{zero} \mid \text{suc } e \mid v \mid \lambda v : \tau . e \mid e\, e \mid \text{match } e\ e\ (v, e) \mid \mu v : \tau . e
\end{aligned}$$

We use the metavariable v to range over variable names, τ to range over types and e to range over expressions. We use *expressions* and *terms* interchangeably. $\lambda_{\rightarrow}$'s type system is described by the following set of rules:

$$\frac{}{\Gamma \vdash \text{zero} : \mathbb{N}}\ \{znat\} \qquad \frac{\Gamma \vdash e : \mathbb{N}}{\Gamma \vdash \text{suc } e : \mathbb{N}}\ \{snat\}$$

$$\frac{\Gamma, v : \tau \vdash e : \tau'}{\Gamma \vdash (\lambda v : \tau . e) : \tau \rightarrow \tau'}\ \{lam\} \qquad \frac{v : \tau \in \Gamma}{\Gamma \vdash v : \tau}\ \{var\}$$

$$\frac{\Gamma \vdash e : \tau \rightarrow \tau' \quad \Gamma \vdash e' : \tau}{\Gamma \vdash e\, e' : \tau'}\ \{app\} \qquad \frac{\Gamma, v : \tau \vdash e : \tau}{\Gamma \vdash (\mu v : \tau . e) : \tau}\ \{rec\}$$

$$\frac{\Gamma \vdash e_1 : \mathbb{N} \quad \Gamma \vdash e_2 : \tau \quad \Gamma, v : \mathbb{N} \vdash e_3 : \tau}{\Gamma \vdash \text{match } e_1\ e_2\ (v, e_3) : \tau}\ \{match\}$$

Rule *znat* types zero as a natural number in any context. Rule *snat* types suc as a natural number as long as its argument is also a natural number. Rule *var* types any element of the context with its typing information. Rules *lam* and *app* type abstractions and applications. Rule *match* requires its first argument to be a natural number and the others to have the same type. The third argument expression is bound to a variable name intended to refer to the predecessor of the first argument, if it is not zero. Finally, rule *rec* types recursive expressions (fixpoint operator).

The semantic style considered in this work is the small step, *call by value* semantics following [23]. In the text, capture-avoiding substitution of variable x by term y in term e is represented by $[x \mapsto y]e$.

3 Expansion and Transformation

Transformation of recursive functions into terminating versions is done considering the two core calculi we defined: System R, which is a subset of the presented λ-calculus also featuring some form of recursion; and the strongly normalizing System L. Translating programs from System R to System L implies recursion elimination. In order to still be able to perform non trivial computations, System R's expressions are expanded before the translation, unrolling the recursive definitions up to a factor we called *fuel* [4]. In this way, the output functions still computes something identical to the original function, although they can never have the same exact semantics, since one is recursive and the other has no automated repetition at all.

System R's syntax establishes two levels for expressions, reserving the top level for recursive definitions and applications only. The bottom level is constituted of the same constructs seen in λ-calculus, except for the μ operator. We made this choice to prevent nested fixpoint expressions. For example, we want to avoid expressions of the form $\mu v_1 : \tau.(\mu v_2 : \tau.e)$. In functional languages such as Haskell and Racket, the programmer is able to define several functions before defining a main expression. Those definitions can be compiled into derived forms, abstracting the main expression and applying it to the function definitions. The semantics of those applications will create nested μ functions. As a syntatic transformation, our unrolling is intended to happen before semantic evaluation. Our choice to allow recursive functions only at top level is just a matter of presentation: we can allow several definitions to be unrolled separately with some minor tweaks (by using a `let in` construction and several fuel values, for example). Once semantic evaluation nests them, they will be already expanded. With some extra effort, the core ideas in this paper should apply to the traditional $\lambda_{\rightarrow}$as well, although it is a conjecture at this point.

[4] The term "fuel" is inspired by Petrol Semantics. It is presented, for instance, in [16].

Recursive definitions also require a functional type, so trivial loops of type $\mathbb{N}$ cannot be expressed (for example, the term $\mu v : \mathbb{N}.v$). The following context-free grammar describes System R's syntax:

$$\begin{aligned} \tau &::= \mathbb{N} \mid \tau \to \tau \\ p &::= \mu v : \tau \to \tau.e \mid p\, e \\ e &::= \text{zero} \mid \text{suc } e \mid v \mid \lambda v : \tau.e \mid e\, e \mid \text{match } e\ e\ (v,\, e) \end{aligned}$$

Meta-variable p ranges over top-level expressions. For convenience, we assume every program will contain a recursive definition, regardless if it is involved in a application or not. Typing rules remain the same as $\lambda_{\to}$ except for *rec* rule that now requires the functional type. The new *rec* rule is:

$$\frac{\Gamma, v : \tau \to \tau' \vdash e : \tau \to \tau'}{\Gamma \vdash (\mu v : \tau \to \tau'.e) : \tau}\ \{rec\}$$

Alternatively, the typing rule for recursive functions in System R could require the expression to have an ocurrence of the variable that was added to the context. This is convenient for proving theorems but is burdensome for random program generation and real life implementations. Variable ocurrence are inductively defined following Definition 1.

Definition 1 (Variable ocurrence). *A variable $v : \tau$ occurs in some term e, denoted by $v : \tau \in_v e$, if e is v or if $v : \tau$ occurs in any of the subterms of e. Otherwise, $v : \tau$ does not occur in e, denoted by $v : \tau \notin_v e$.*

Informally, the semantics of System R is the same of $\lambda_{\to}$. But since System R's syntax won't allow a fixpoint inside a bottom-level term, the conventional evaluation rule for recursive functions cannot be applied. The solution is to elaborate System R to $\lambda_{\to}$, which is straightforward, given System R programs are a subset of $\lambda_{\to}$. In this way, only during evaluation recursive definitions are allowed to bypass the syntactic restrictions. Notice that there is no need to modify the rule to reflect the constraint of fixpoints having functional type, because that is just a particular case of the more general rule already present in $\lambda_{\to}$.

3.1 Unrolling

The inlining of a function f means to replace the ocurrences of f for its body, and is largely used by developers of compilers to increase performance of their programs [6]. By inlining recursive definitions into themselves, we create an equivalent expression that performs fewer recursive calls. As Rugina and Rinard [21] point out, we need to be careful to not super-exponentially grow the code of the functions. For example, consider this pseudo-Haskell function for adding two numbers in Peano notation:

```
sum :: Nat → Nat → Nat
sum = \x y → case x of
   Zero   → y
   Suc w → sum w (Suc y)
```

The idea in this expression is to match the first number x over the patterns zero or suc. If it is zero, we simply answer with the second number y. Otherwise we name w the predecessor of x and recursively call the function over w and the successor of y. Inlining this function with itself once would give us the following expression:

```
sum :: Nat → Nat → Nat
sum = \x y → case x of
  Zero   → y
  Suc w → (\x y → case x of
    Zero   → y
    Suc w → sum w (Suc y)) w (Suc y)
```

We had one match (`case of`, in Haskell) before and now we have two. If we inline this last expression with itself we would end up with four matches, because the definition of *sum* now contains two of them:

```
sum :: Nat → Nat → Nat
sum = \x y → case x of
 Zero   → y
 Suc w → (\x y → case x of
  Zero   → y
  Suc w → (\x y → case x of
   Zero   → y
   Suc w → (\x y → case x of
    Zero   → y
    Suc w → sum w (Suc y)) w (Suc y)) w (Suc y)) w (Suc y)
```

Doubling the number of matchings every step in a simple term like this is not in our best interest. Instead, we want to increase these matches just by one in each step. Inlining the original expression 2 times should actually give us the following term:

```
sum :: Nat → Nat → Nat
sum = \x y → case x of
  Zero   → y
  Suc w → (\x y → case x of
    Zero   → y
    Suc w → (\x y → case x of
      Zero   → y
      Suc w → sum w (Suc y)) w (Suc y)) w (Suc y)
```

If the original function had two recursive calls, then we would want to increase the number of matches by two in each step, and that would still be better than having 2^{m+1} matches after m steps. So, we have to always keep track of the original expression and inline its body inside the output expressions. Definition 2 formalizes this behavior.

Definition 2. *The n-expansion of a System R expression e is the result of the nth cumulative inlining of e, described by exp in the following algorithm:*

$$
\begin{aligned}
inl(v, v', e) &= \begin{cases} e & \text{if } v \equiv v' \\ v & \text{otherwise} \end{cases} \\
inl(zero, v', e) &= zero \\
inl(suc\ e', v', e) &= suc\ (inl(e', v', e)) \\
inl(e_1\ e_2, v', e) &= (inl(e_1, v', e))\ (inl(e_2, v', e)) \\
inl(\lambda v : \tau.e', v', e) &= \lambda v : \tau.(inl(e', v', e)) \\
inl(match\ e_1\ e_2\ (w, e_3), v', e) &= match\ (inl(e_1, v', e))\ (inl(e_2, v', e)) \\
&\quad (w, (inl(e_3, v', e)))
\end{aligned}
$$

$$
\begin{aligned}
exp'(e', \_, \_, 0) &= e' \\
exp'(e', v, e, n) &= inl(exp'(e', v, e, n-1), v, e)
\end{aligned}
$$

$$
\begin{aligned}
exp(\mu v : \tau \to \tau'.e, n) &= \mu v : \tau \to \tau'.(exp'(e, v, e, n)) \\
exp(p\ e, n) &= (exp(p, n))\ e
\end{aligned}
$$

Function $inl(e', v, e)$ performs the inlining of e inside e' replacing the ocurrences of v. This is only allowed between bottom-level expressions, so the fixpoint construction can never appear here. Function $exp'(e', v, e, n)$ accumulates the inlinings n times, always using the original e instead of previous result with itself. Function $exp(e, n)$ expands a recursive function v by n-expanding its body replacing the ocurrences of v. Functions inl, exp' and exp halt for every input (Lemmas 1 and 2) and the resulting term is still a recursive function (Lemma 3), whose recursive calls must be now eliminated.

Lemma 1 (Inlining Halts). *For all bottom-level terms e, e' of System R, and every variable v, there exists a bottom-level term e'' such that $inl(e', v, e) = e''$.*

Proof. By induction on the structure of e. Base cases for variable and zero immediately produce a term. In the other cases inlining is merely propagated to subterms, which are strictly smaller. □

Lemma 2 (Expansion Halts). *For every $n \in \mathbb{N}$ and System R expression e, there exists an expression e' such that $exp(e) = e'$.*

Proof. The expansion of bottom-level terms, done in the exp' function, always halts because inlining halts and n is always decreasing towards 0, in which case the expression is left unchanged. Top-level expressions come in two cases.

- Case 1: $e = \mu v : \tau \to \tau'.e_b$. Its body e_b, which is a bottom-level expression, is expanded.

– Case 2: $e = (p)\,(e_r)$. The top-level p is expanded with the same n, bottom-level e_r is left unchanged. By the structure of the grammar, a top level expression will always end with a recursive definition, falling in case 1.

□

Lemma 3 (Occurrence Preserving Expansion). *For every $n \in \mathbb{N}$, term e and variable $v : \tau$, if $v : \tau \in_v e$ then $v : \tau \in_v exp(e, n)$.*

Proof. Inlining substitutes all ocurrences of a variable v_1 by some e_1 inside some e_2. If v_1 occurs in e_1 and in e_2, then by induction, v_1 occurs in $inl(e_2, v_1, e_1)$. The expansion of a recursive definition $\mu v : \tau \to \tau'.e$ inlines e within e replacing v in every step. So if v occurs in e then it follows that v still occurs after each cumulative inlining, and thus after the expansion. □

3.2 Recursion Elimination

System L's syntax offers no way to express any kind of recursion, but offers a construction for abnormal termination, namely the term `error`:

$$\begin{aligned} \tau &::= \mathbb{N} \mid \tau \to \tau \\ e &::= \text{zero} \mid \text{suc } e \mid v \mid \lambda v : \tau.e \mid e\ e \mid \text{match } e\ e\ (v,\ e) \mid \text{error} \end{aligned}$$

Its type system is identical to $\lambda_\to$ minus the rule for recursion, and adding the rule for typing error with an arbitrary type:

$$\frac{\forall \tau}{\Gamma \vdash \text{error } : \tau}\ \{error\}$$

The semantics for System L is the same as $\lambda_\to$'s but including three rules for propagating errors and removing the $\{rec\}$ rule. The error construction, although a normal form, is not considered a value. In this way, the semantics remains deterministic [20]. The included rules are:

$$\frac{}{\text{suc error} \longrightarrow \text{error}}\ \{esuc\} \qquad \frac{}{\text{error } e_2 \longrightarrow \text{error}}\ \{eapp_1\}$$

$$\frac{}{v_1 \text{ error} \longrightarrow \text{error}}\ \{eapp_2\} \qquad \frac{}{\text{match error } e_1\ e_2\ (v, e_3) \longrightarrow \text{error}}\ \{ematch\}$$

Since System L has no way of expressing recursion or any form of repetition there is no way a program in System L loops forever. Translating a term from System R to System L will produce a program that will surely terminate. To achieve this, we must eliminate the remaining recursive calls, i.e., the occurrences of the function name in the expression. Definitions 3 and 4 formally describe recursion elimination and translation.

Definition 3. *Ocurrence elimination of a variable v from a bottom-level term e is the translation of e to System L, replacing v ocurrences by error. It is defined* *by this algorithm:*

$$\begin{aligned}
elim(v', v) &= \begin{cases} error & if\ v \equiv v' \\ v' & otherwise \end{cases} \\
elim(zero, v) &= zero \\
elim(suc\ e, v) &= suc\ (elim(e, v)) \\
elim(\lambda v' : \tau.e, v) &= \lambda v' : \tau.(elim(e, v)) \\
elim(e_1\ e_2, v) &= (elim(e_1, v))\ (elim(e_2, v)) \\
elim(match\ e_1\ e_2\ (v', e_3), v) &= match\ (elim(e_1, v))\ (elim(e_2, v)) \\
&\quad (v', elim(e_3, v))
\end{aligned}$$

Definition 4. *The translation of a System R expression e is the ocurrence elimination of the recursive function's name from its own body:*

$$\begin{aligned}
tsl(\mu v : \tau \to \tau'.e) &= elim(e, v) \\
tsl(p\ e) &= (tsl(p))\ e
\end{aligned}$$

The second case of *tsl* leaves the right term of application unchanged, since bottom-level System R expressions are a subset of System L. In an actual implementation, an auxiliary function is necessary to simply translate bottom-level terms into their System L's counterparts. The composition of unrolling and translation gives us the transformation algorithm, which always halts (Theorem 1):

$$transform(e, f) = tsl\,(exp(e, f))$$

Lemma 4 (Ocurrence Elimination Halts). *For every bottom-level System R term e and variable v, there exists a System L expression e' such that $elim(e, v) = e'$.*

Proof. Ocurrence elimination behaves similar to inlining. In inlining, the ocurrences of v are replaced by some term e_1. In occurence elimination, the ocurrences of v are replaced by the term *error*. By induction, if the term is a variable or a zero, it is immediately translated by error or its counterpart in System L and then halts. If not, the ocurrence elimination is propagated to the subterms, which are strictly smaller. □

Theorem 1 (Transformation Halts). *For every $f \in \mathbb{N}$ and System R expression e, there exists a System L expression e' such that $transform(e, f) = e'$.*

Proof. Translation erases the μ construction, which is absent in System L, and then eliminates the ocurrences of the recursive function's name from the body. Transformation composes n-expansion and translation. Since both functions always halt, their composition always halts too. □

In the example program for adding two numbers x and y, when the ocurrences of the function's name are eliminated in the translating process, the resulting expression can still answer with y if x is 0: $\mu s : \mathbb{N} \to \mathbb{N} \to \mathbb{N}.\lambda x : \mathbb{N}.\lambda y : \mathbb{N}.\text{match}\ x\ y\ (w, (\text{error}\ w)\ (\text{suc}\ y))$.

If we expand the expression f times before translating, the System L version would be able to successfully compute the sum whenever $x \leq f$, reducing to error otherwise. There can be no general way of guessing f, since it could solve the halting problem. One disadvantage of our approach is to rely on the programmer to inform the fuel for the function, similar to what they have to do when using bounded loops in imperative languages. One advantage of this approach is that some infinite loops are immediately reduced to error. No matter how high is f, if we transform $\mu v : \mathbb{N} \to \mathbb{N}.v$ the output program will always be one simple error.

4 Term Generation

Property-based testing is an interesting approach to testing, because it relies on generating random programs to check properties, avoiding the bias when developers create manual test cases. But using this approach when checking properties of compilers is not easy, because it is necessary to define how to generate valid programs of that language. In this work, since we are directly dealing with termination, we are also interested in generating terminating programs.

To accomplish this, we define for our generation procedure a superset of types, in which our base type $\mathbb{N}$ is indexed by natural numbers. Those indexes are related via subtyping, such that $\mathbb{N}^x <: \mathbb{N}^y$ whenever $x \leq y$. Functions are related in the usual way, being covariant in return type and contravariant in the argument type. Our generation judgment $\Gamma; d; r; \tau \rightsquigarrow e$ means that expression e of type τ can be generated from context Γ, given limits d for expression depth and r for type indexes. The relation $\rightsquigarrow$ is annotated with a letter to distinguish the form of the expressions it generates. $\xi(xs)$ selects a random element from a non-empty list xs. Our first rules are the generation of zero and terms that are variables. A zero can be generated in every scenario where a $\mathbb{N}$ is expected, regardless of the index. Variables can be picked from a non empty list of candidates. Those candidates are the variables from the context that are a subtype of the expected type.

$$\frac{}{\Gamma; d; r; \mathbb{N}^x \rightsquigarrow_z zero}\ \{zero\} \qquad \frac{cs = \{v \mid v : \tau' \in \Gamma \land \tau' <: \tau\} \quad cs \neq \emptyset}{\Gamma; d; r; \tau \rightsquigarrow_v \xi(cs)}\ \{var\}$$

The successor construction can be also generated in every scenario where a $\mathbb{N}^x$ is expected, as long as index x is not already 0. Its argument can be zero, a variable or another successor construction. The index limit is decreased for the generation of its subterm. When we generate pattern matchings inside recursive definitions, the first branch must be a constant, so it is useful to not include the other rules as possible subterms for this construction. The notation $\psi, \phi, \dots = \xi(\{a, b, c, d, \dots\})$ means that rules annotated with ψ, ϕ, etc., are an alias to rules annotated with any letter in the list inside ξ, and are meant to be selected randomly for each letter on the left side.

$$\frac{\Gamma; d; r; \mathbb{N}^x \rightsquigarrow_\psi e \quad \psi = \xi(\{z, v, s\})}{\Gamma; d; r+1; \mathbb{N}^{x+1} \rightsquigarrow_s \mathit{suc}\, e} \{suc\}$$

Abstractions need to generate an expression of the return type before inserting a λ. The function *fresh* generates a variable name that is not present in the given context. Operator $\lfloor\tau\rfloor$ erases index from indexed type τ.

$$\frac{v = \mathit{fresh}(\Gamma) \quad \Gamma, v : \tau_1; d; r; \tau_2 \rightsquigarrow_\psi e \quad \psi = \xi(\{z, v, s, a\})}{\Gamma; d; r; \tau_1 \to \tau_2 \rightsquigarrow_a \lambda v : \lfloor\tau_1\rfloor.e} \{abs\}$$

Applications of type τ need the generation of a function that results in that τ, applied to an appropriate argument, which has also to be generated. Operator Θ generates a type given the limits d and r, and never associates to the left. This operator is defined by randomly selecting one out of two more specific versions: $\Theta^{\to}$ that generates only function types and $\Theta^{\mathbb{N}}$ that generates only indexed $\mathbb{N}$ types.

$$\begin{aligned}
\Theta^{\mathbb{N}}(r) &= \mathbb{N}^{\xi(\{1,\ldots,r\})} \\
\Theta^{\to}(0, r) &= (\Theta^{\mathbb{N}}(r)) \to (\Theta^{\mathbb{N}}(r)) \\
\Theta^{\to}(2d, r) &= (\Theta^{\mathbb{N}}(r)) \to (\xi(\{\Theta^{\mathbb{N}}(r), \Theta^{\to}(d, r)\})) \\
\Theta(d, r) &= \xi(\{\Theta^{\mathbb{N}}(r), \Theta^{\to}(d, r)\})
\end{aligned}$$

If limits are high enough, both base and functional types can be generated. Otherwise, it will force the generation of a base type ($\mathbb{N}$ is our only base type here). Generating applications will cut the d limit to half for generating its subterms.

$$\frac{\tau' = \Theta(d, r) \quad \Gamma; d; r; \tau' \to \tau \rightsquigarrow_\phi e_1 \quad \Gamma; d; r; \tau' \rightsquigarrow_\psi e_2 \quad \phi, \psi = \xi(\{z, v, s, a\})}{\Gamma; 2d; r; \tau \rightsquigarrow_a e_1\, e_2} \{app\}$$

Match constructions need to generate a term of type $\mathbb{N}$ and two terms of the expected type. For the third term e_3, a fresh variable is added to context with a decreased index limit, and this limit is decreased for e_3 as well. The operator $_\downarrow$ decreases the index of the type. If it is a function, only the leftmost atom is decreased. Matches can only be introduced when limit d is greater than 1.

$$\frac{\begin{array}{ll} v = \mathit{fresh}(\Gamma) & \Gamma; d; r; \mathbb{N}^r \rightsquigarrow_\phi e_1 \\ \phi, \psi, \rho = \xi(\{z, v, s, a\}) & \Gamma; d; r; \tau \rightsquigarrow_\psi e_2 \quad \Gamma, v : \mathbb{N}^{r_\downarrow}; d; r_\downarrow; \tau \rightsquigarrow_\rho e_3 \end{array}}{\Gamma; 2d; r; \tau \rightsquigarrow_a \mathit{match}\, e_1\, e_2\, (v, e_3)} \{match\}$$

The generation of recursive functions and applications involving them require more caution. Generating a recursive function needs the generation of a proper body, which means it has to be well-typed and must terminate.

$$\frac{v = \mathit{fresh}(\Gamma) \quad \Gamma, v : (\tau_1 \to \tau_2)_\downarrow; d; r_\downarrow; \tau_1 \to \tau_2 \rightsquigarrow_b e}{\Gamma; 2d; r+1; \tau_1 \to \tau_2 \rightsquigarrow_f \mathit{rec}\, v : \lfloor\tau_1 \to \tau_2\rfloor.e} \{rec\}$$

The simplest way of guaranteeing this is inserting λs until we are left with the generation of a $\mathbb{N}$, and then generate a match. This match will generate an immediate natural number on its first branch and it will allow a recursive call on the second. The expression being matched needs to be zero or a variable. The expression being generated for the second branch needs to be standardized, i.e., modified to make sure that, if there is a recursive call, then its first argument will be the predecessor of the matching expression.

$$\frac{\begin{array}{lll} v = \mathit{fresh}(\Gamma) & & \\ \rho = \xi(\{z, v, s, a\}) & \Gamma; d; r; (\mathbb{N}^x)_\downarrow \rightsquigarrow_\psi e_1 & \Gamma, v : (\mathbb{N}^x)\downarrow; d; r; \mathbb{N}^x \rightsquigarrow_\rho e_3' \\ \phi, \psi = \xi(\{z, v, s\}) & \Gamma; d; r; \mathbb{N}^x \rightsquigarrow_\phi e_2 & e_3 = \mathit{stdz}(e_3'; \Gamma, v : (\mathbb{N}^x)_\downarrow) \end{array}}{\Gamma; d; r; \mathbb{N}^x \rightsquigarrow_b \mathit{match}\, e_1\, e_2\, (v, e_3)} \{\mathit{buildBody1}\}$$

$$\frac{v = \mathit{fresh}(\Gamma) \quad \Gamma, v : \tau_1; d; r; \tau_2 \rightsquigarrow_b e}{\Gamma; d; r; \tau_1 \to \tau_2 \rightsquigarrow_b \lambda v : \lfloor \tau_1 \rfloor . e} \{\mathit{buildBody2}\}$$

The standardization process is defined by the following algorithm, where bottom is a function that returns the first variable name added to the context, i.e., the bottom of the scope stack; and top is a function that returns the last variable name added to the context, i.e., the top of the scope stack.

$$\begin{aligned} \mathit{std}(e_1\, e_2, v_1, v_2) &= \begin{cases} e_1\, v_2 & e_1 \equiv v_1 \\ (\mathit{std}(e_1, v_1, v_2))\, (\mathit{std}(e_2, v_1, v_2)) & \mathit{otherwise} \end{cases} \\ \mathit{std}(\lambda v : \tau . e, v_1, v_2) &= \lambda v : \tau . (\mathit{std}(e, v_1, v_2)) \\ \mathit{std}(\mathit{match}\, e_1\, e_2\, (v, e_3), v_1, v_2) &= \mathit{match}\, (\mathit{std}(e_1, v_1, v_2))\, (\mathit{std}(e_2, v_1, v_2)) \\ & \quad\; (v, \mathit{std}(e_3, v_1, v_2)) \\ \mathit{std}(e, \_, \_) &= e \end{aligned}$$

$$\mathit{stdz}(e, \Gamma) = \mathit{std}(e, \mathit{bottom}(\Gamma), \mathit{top}(\Gamma))$$

Applications involving recursive definitions are most useful if they yield a number as a final value, and so we force this to happen. After generating a functional type and a recursive definition of this type, we generate proper arguments and apply them to the function.

$$\frac{\tau = \Theta^{\to}(d, r) \quad \Gamma; d; r; \tau \rightsquigarrow_f e' \quad \Gamma; d; r; e'; \mathit{revargs}(\tau) \rightsquigarrow_c e}{\Gamma; 2d; r; \mathbb{N}^x \rightsquigarrow_g e} \{\mathit{apprec}\}$$

$$\begin{aligned} \mathit{args}(\mathbb{N}^x) &= [] \\ \mathit{args}(\mathbb{N}^x \to \tau_2) &= \mathbb{N}^x :: (\mathit{args}(\tau_2)) \\ \mathit{revargs} &= \mathit{reverse} \circ \mathit{args} \end{aligned}$$

For this, we build a list of arguments and extend the judgement to contain them. We must build the applications of the outer arguments first, applying the recursive definition to the first argument as the innermost application. For this, we need to assume our list of arguments is reversed.

$$\frac{\begin{array}{c}\psi = \xi(\{z, v, s, a\}) \\ \Gamma; d; r; \mathbb{N}^x \ \rightsquigarrow_\psi \ e'\end{array}}{\Gamma; d; r; e; [\mathbb{N}^x] \rightsquigarrow_c e\, e'}\ \{bdA1\} \qquad \frac{\begin{array}{cc} & \psi = \xi(\{z, v, s, a\}) \\ \Gamma; d; r; e; ts \rightsquigarrow_c e_1 & \Gamma; d; r; \mathbb{N}^x \ \rightsquigarrow_\psi \ e_2\end{array}}{\Gamma; d; r; e; \mathbb{N}^x :: ts \rightsquigarrow_c e_1\, e_2}\ \{bdA2\}$$

Finally, generating a System R expression means to generate a type and generate a term accordingly:

$$\frac{\Gamma; d; r; \mathbb{N}^x \rightsquigarrow_\psi e}{\Gamma; d; r; \mathbb{N}^x \rightsquigarrow e} \quad \frac{\Gamma; d; r; \tau_1 \to \tau_2 \rightsquigarrow_\phi e}{\Gamma; d; r; \tau_1 \to \tau_2 \rightsquigarrow e}$$

If we want only top terms of System R, then it is enough to use $\psi = g$ and $\phi = f$. But it is interesting to allow the generation of bottom level terms when testing properties, since recursive functions are optional in real-life implementations. In that case, $\psi = \xi(\{z, v, s, a, g\})$ and $\phi = \xi(\{a, f\})$.

4.1 Soundness of Term Generation

Lemma 5 (Bottom level generation is sound). *For every context Γ, natural numbers d and r, type τ and $\psi \in \{a, s, z, v\}$, if $\Gamma; d; r; \tau \rightsquigarrow_\psi e$ then $e : \tau$.*

Proof. The rule for {zero} can only be used to generate terms of type $\mathbb{N}$, which is the type of zero and the rule for {var} pick a value of type τ' from the context if available, where τ' and τ are equal up to erasure of indices. Rules {suc}, {abs} and {match} directly represent their counterparts in the type system, generating subterms of the appropriate type. The rule for {app} generates a random type to construct a functional type with return of type τ and then generates both the function and its argument, building an application with them. □

Lemma 6 (Bottom level generation halts). *For every context Γ, natural numbers $d \geq 2$ and r, and type τ, there exists a term e such that $\Gamma; d; r; \tau \rightsquigarrow_a e$.*

Proof. Type τ is either $\mathbb{N}$ or a function type. The first case has rules {match} and {app} available (since $d \geq 2$) and the other is generated by {abs}. When generating subterms for {match}, {abs} or {app}, the rules {zero}, {var} and {suc} can be used. Rules {zero} and {var} trivially halt. Rule {suc} decreases r and the index of $\mathbb{N}$ by 1 for the generation of its subterms and cannot be used when those values are 0. The subterm generated by rule {abs} has a type strictly smaller then τ and it cannot be used once τ becomes $\mathbb{N}$. Rules {app} and {match} decrease the d limit by half for the generation of their subterms, and cannot be used when $d < 2$. Therefore, some value in the triple (d, r, τ) will always decrease for the generation of a subterm. The rule {zero} can be used to generate a trivial term whenever rules {app}, {match} and {abs} cannot be used anymore and there are no candidates for using {var}. □

Theorem 2 (Top level generation is sound). *For every context Γ, natural numbers d and r, type τ and $\psi \in \{f, g\}$, if $\Gamma; d; r; \tau \rightsquigarrow_\psi e$ then $e : \tau$.*

Proof. If τ is a functional type, e is generated using {rec}. This rule creates a fresh var of type τ and adds it to the context when generating a subterm of type τ, satisfying the corresponding typing rule. The subterm is generated using a special procedure that builds an adequate body for the recursive function. This procedure generates well-typed λ abstractions until τ is a single $\mathbb{N}$, generating a well-typed pattern matching right away. If τ is already $\mathbb{N}$, a functional type and a recursive function p of this type are generated by rule {apprec}, then the type of each argument is used to generate bottom-level values and build nested applications until p has all arguments necessary to generate a $\mathbb{N}$. $\square$

Theorem 3 (Top level generation halts). *For every context Γ, natural numbers $d \geq 2$ and r, and type τ, there exists a term e such that either $\Gamma; d; r; \tau \rightsquigarrow_f e$ or $\Gamma; d; r; \tau \rightsquigarrow_g e$.*

Proof. If τ is a functional type, it must follow that $\Gamma; d; r; \tau \rightsquigarrow_f e$. Rule {rec} depends on $\rightsquigarrow_b$ for generating its subterm. By using {buildBody2}, τ is decreasing for the generation of the subterms of the λ inserted abstractions. When it reaches $\mathbb{N}$, bottom level terms are generated for the insertion of the match. If τ is already $\mathbb{N}$, bottom level terms are generated for each argument type of the generated functional type, finishing with the application of the recursive function when the list of arguments contains only the last one. $\square$

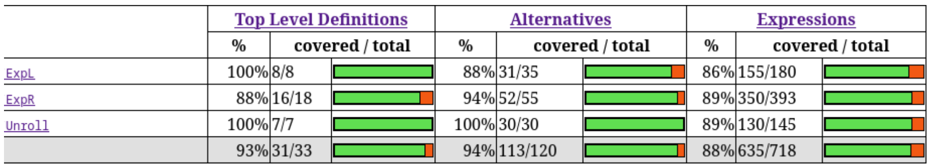
5 Quick-checking Properties

We implemented Ringell [10], an interpreter for both System R and System L. When given only a source-code file, it will parse the text, typecheck the abstract syntax tree, use System R as the internal representation and run it using the $\lambda_{\rightarrow}$interpreter as seen in [23]. When given a source-code file and a nonnegative integer number n, it will transform the System R representation into System L, using n as fuel for expansion, and then run it using the $\lambda_{\rightarrow}$interpreter as seen in [20]. Ringell's syntax, inspired in Haskell and $\lambda_{\rightarrow}$, is simple and support just a few syntactic constructions, making it close to both System R and System L syntax.

The term generation procedure is implemented in Ringell's test modules. We specified some properties using Quick Check [19], invoking modules from Ringell for validation. The properties are as follows:

- Property 1: All generated System R programs are well typed
- Property 2: All generated System R programs terminate
- Property 3: For every generated System R term e of type $\mathbb{N}$, if e terminates with value v doing at most f recursive calls, then *transform*(e, f) yields v.
- Property 4: For every generated System R term e of type $\mathbb{N}$ and some fuel f, if *transform*(e, f) yields a value v, then e terminates resulting in v.

Property 2 is only true because of our generation procedure, it is obviously not a property of System R. Properties 3 and 4 are our desired semantic properties for the transformation technique: the output function results in the same value of the original if enough fuel is given and the output function yielding a value implies the original function terminates with the same value. Both properties concern only programs of the base type ℕ, because there are two syntatic constructors for programs of functional type in System R and only one in System L, and we prefer using propositional equality instead of creating a more complex equivalence relation for this purpose. Figure 1 presents the coverage results generated by Haskell Stack coverage flag, after running thousands of successful tests for each property.



	Top Level Definitions			Alternatives			Expressions		
	%	covered / total		%	covered / total		%	covered / total	
ExpL	100%	8/8	🟩	88%	31/35	🟩🟥	86%	155/180	🟩🟥
ExpR	88%	16/18	🟩🟥	94%	52/55	🟩🟥	89%	350/393	🟩🟥
Unroll	100%	7/7	🟩	100%	30/30	🟩	89%	130/145	🟩🟥
	93%	31/33	🟩🟥	94%	113/120	🟩🟥	88%	635/718	🟩🟥

Fig. 1. Test Coverage Results

The first column are for modules names, in which `ExpR` and `ExpL` are the interpreters and `Unroll` is where the functions involved in transformation are. Most of non-reached code concerns error control and arguments of functions that were not used in all cases of the function (for example, context is not needed for evaluating a `zero`).

6 Related Work

There are some papers closely related to ensuring termination of functional programs, although the systems they describe do not focus on syntactically transforming the programs.

Abel's foetus [1,2,3,4] is a simplification of Munich Type Theory Implementation, and features a termination checker for simple functional programs. It builds an hypergraph of function calls, computes its transitive closure and tries to find a decreasing lexicographic order in the recursive functions' arguments. Foetus is pretty limited, forcing the programmer to construct several auxiliary, often mutually recursive, functions to convince the checker.

Barthe's Type based termination [5,8,9] is a strategy in which the language's types carries more information so the type system can serve as a proof system that the programs terminate. It is an elegant but complicated approach, and since it does not transform the programs syntactically, targets with stringent security requirements would still not accept the functions.

Also, LLVM's Clang [15] offers a macro to unroll bounded loops in languages of C family, which is largely used when using pseudo-C code to compile for eBPF. GCC [12] has a command line option to unroll loops in C programs. The features in both compilers are unable to deal with recursion. At the time of this paper, Microsoft's MSVC [17] has no way to directly instruct the compiler to unroll a loop.

Finally, our generation procedure and test approach are based on the work in [11]. It formalizes a type-directed algorithm to generate random programs of Featherweight Java, which is used to verify several properties using QuickCheck.

7 Conclusion

It is quite useful to allow the dynamic change of behavior in computing systems. This level of adaptation is essential to complete several tasks with efficiency, being attractive even for systems with stringent security requirements, such as the Linux kernel. Because infinite loops can bring a whole system down, a very common restriction in those systems is that the programs must terminate, which is impossible to check in general. Although imperative languages have bounded loops to walk around this limitation, functional languages are left empty-handed. In this work, we proposed a technique to unroll recursive programs in functional languages. For this task, we defined two core languages: one with recursion (System R) and one without recursion but with syntax for errors (System L). We defined an expansion algorithm for System R and a translation algorithm from System R to System L. Our strategy is guaranteed to terminate, and can be used to create compilers from general purpose functional languages to restricted scenarios such as eBPF or smart contracts for blockchain networks. Our prototype implementation of the strategy, Ringell, is available to public access.

References

1. Abel, A.: foetus — termination checker for simple functional programs. Tech. rep., Ludwigs-Maximilians-University, Munich (1998)
2. Abel, A.: Specification and verification of a formal system for structurally recursive functions. In: International Workshop on Types for Proofs and Programs. pp. 1–20. Springer, Berlin (1999)
3. Abel, A., Altenkirch, T.: A semantical analysis of structural recursion. In: Fourth International Workshop on Termination WST. pp. 24–25. Darmstadt University of Technology, Dagstuhl, Germany (1999)
4. Abel, A., Altenkirch, T.: A predicative analysis of structural recursion. Journal of Functional Programming **12**(1), 1–41 (2002)
5. et al., G.B.: Type-based termination of recursive definitions. Mathematical structures in computer science **14**(1), 97–141 (2004)
6. Appel, A.: Modern Compiler Implementation in ML. Cambridge University Press, New York (2004)
7. Authors of eBPF: eBPF - Introduction, Tutorials and Community Resources. https://ebpf.io/ (2022)
8. Barthe, G., Grégoire, B., Riba, C.: A tutorial on type-based termination. In: International LerNet ALFA Summer School on Language Engineering and Rigorous Software Development. pp. 100–52. Springer, Piriapolis, Uruguay (2008)

9. Barthe, G., Grégoire, B., Riba, C.: Type-based termination with sized products. In: International Workshop on Computer Science Logic. pp. 493–507. Springer, Berlin (2008)
10. Developers, R.: Ringell. https://github.com/mayconamaro/ringell (2022)
11. Feitosa, S., Ribeiro, R., Du Bois, A.: A type-directed algorithm to generate random well-typed Java 8 programs. Science of Computer Programming **196**, 102494 (2020)
12. GNU: GCC, the GNU Compiler Collection. https://gcc.gnu.org/ (2022)
13. Huet, G., Kahn, G., Paulin-Mohring, C.: The Coq proof assistant a tutorial. Rapport Technique **178** (1997)
14. Le, T.C., Xu, L., Chen, L., Shi, W.: Proving conditional termination for smart contracts. In: Proceedings of the 2nd ACM Workshop on Blockchains, Cryptocurrencies, and Contracts. pp. 57—59. BCC '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3205230.3205239, https://doi.org/10.1145/3205230.3205239
15. LLVM Project: Clang C Language Family Frontend for LLVM. https://clang.llvm.org/ (2022)
16. McBride, C.: Turing-completeness totally free. In: International Conference on Mathematics of Program Construction. pp. 257–275. Springer, Könnigswinter, Germany (2015)
17. Microsoft: Visual Studio Compiler for Windows. https://visualstudio.microsoft.com/cplusplus/ (2022)
18. Norell, U.: Towards a practical programming language based on dependent type theory. Ph.D. thesis, Chalmers University of Technology and Göteborg University, Sweden (2007)
19. O'Sullivan, B., Goerzen, J., Stewart, D.: Real World Haskell. O'Reilly, Sebastopol (2008)
20. Pierce, B.C.: Types and programming languages. MIT press, Cambridge, Massachusetts (2002)
21. Rugina, R., Rinard, M.: Recursion unrolling for divide and conquer programs. In: International Workshop on Languages and Compilers for Parallel Computing. pp. 34–48. Springer, Yorktown, USA (2000)
22. Sipser, M.: Introduction to the Theory of Computation, 3rd edition. Cengage Learning, Cambridge, Massachusetts (2012)
23. Wadler, P., Kokke, W., Siek, J.G.: Programming language foundations in Agda (Jul 2020), http://plfa.inf.ed.ac.uk/20.07/