Simple Programs on Binary Trees – Testing and Decidable Equivalence

Dimitur Krustev

IGE+XAO Balkan, Bulgaria
dkrustev@ige-xao.com

Abstract. We consider a class of simple (iteration/recursion-free) programs operating on unlabeled binary trees. We introduce a cumulative hierarchy of subclasses of programs, which cover the whole class, such that a finite adequate test set exists for each subclass. By taking the minimum subclass, in which a pair of programs live, we can decide just by testing if they are extensionally equivalent.

1 Introduction

Program testing and decidability of program equivalence are two topics with numerous theoretical and practical applications. These topics are closely related on a fundamental level [3]. Both problems – existence of an adequate finite test set and program equivalence – are undecidable not only for Turing-complete languages, but also for many smaller classes of programs [3]. So, discovering classes of programs, for which one or both problems are decidable, can be of great interest. Here we study one such specific class of programs operating on unlabeled binary trees. Programs are composed of operations for building trees, checking tree emptiness, and extracting subtrees. The language is variable-free, similar to Backus' FP [2].

The main results we present are:

- the definition of a cumulative hierarchy of subclasses of programs covering the whole class – Sect. 4.2 (based on a definition of program normal forms introduced previously by the author [8, 10]);
- proofs of existence (Sect. 4.4) and optimality (Sect. 4.5) of finite adequate test sets for programs in each subclass;
- a decision procedure for program equivalence (Sect. 5), based on the existence of finite adequate test sets.

We start by introducing the class of simple programs we consider (Sect. 2). We then briefly review the notion of program normal forms (Sect. 3), which is obtained by program transformations directly inspired by supercompilation and deforestation [13, 15, 16]. Some definitions related to program testing are briefly introduced in Sect. 4.1. The rest of Sect. 4 is devoted to the description of our main result – the existence of finite adequate test sets.

We can illustrate the proposed method on a simple example. Consider the 2 programs in Table 1. The first one (`I`) simply returns the input tree unchanged. The second (`ifnil(I,nil,cons(hd,tl))`) returns an empty tree if the input tree is empty, otherwise it builds a new tree containing the left and right subtree of the input as left and right subtree correspondingly. Clearly both programs compute the identity function. We can prove this fact in many ways, but using the main result of this article (Theorem 1) we can just check it by direct computation – comparing the results of the 2 programs on input trees of depth ≤ 2.

Table 1. Example of deciding program equivalence by testing

input	`I`	`ifnil(I,nil,cons(hd,tl))`
`nil`	`nil`	`nil`
`(nil . nil)`	`(nil . nil)`	`(nil . nil)`
`((nil . nil) . nil)`	`((nil . nil) . nil)`	`((nil . nil) . nil)`
`(nil . (nil . nil))`	`(nil . (nil . nil))`	`(nil . (nil . nil))`
`((nil . nil) . (nil . nil))`	`((nil . nil) . (nil . nil))`	`((nil . nil) . (nil . nil))`

2 Simple Programs on Binary Trees

The programs we consider operate on unlabeled binary trees. We use textual Lisp-like notation for such trees, whose grammar is:

$$T \ ::= \ \texttt{nil} \mid (T \ . \ T)$$

As an example, the notation `(nil . (nil . nil))` corresponds to the tree in Fig. 1.

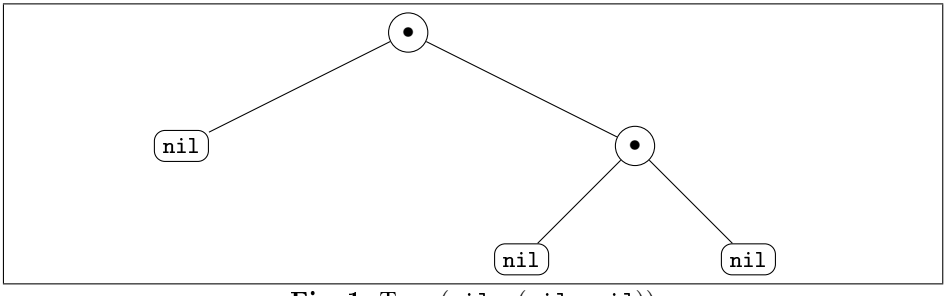

Fig. 1: Tree `(nil . (nil . nil))`

While this data structure is simple, it is universal, as we can encode arbitrary data as such trees. We give examples of such possible encodings below:

– $\lceil \bullet \rceil_{\text{Bool}} \ : \ \text{Bool} \to T$

$$\begin{aligned} \lceil \text{false} \rceil_{\text{Bool}} &= \text{nil} \\ \lceil \text{true} \rceil_{\text{Bool}} &= (\text{nil . nil}) \end{aligned}$$

– $\lceil\bullet\rceil_{\mathbb{N}} : \mathbb{N} \to T$

$$\begin{array}{lcl} \lceil 0\rceil_{\mathbb{N}} & = & \text{nil} \\ \lceil n+1\rceil_{\mathbb{N}} & = & (\text{nil} \mathbin{.} \lceil n\rceil_{\mathbb{N}}) \end{array}$$

– $\lceil\bullet\rceil_{\text{List}(X)} : \text{List}(X) \to T$

$$\begin{array}{lcl} \lceil[]\rceil_{\text{List}(X)} & = & \text{nil} \\ \lceil[x_1, x_2, \ldots, x_n]\rceil_{\text{List}(X)} & = & (\lceil x_1\rceil_X \mathbin{.} \lceil[x_2, \ldots, x_n]\rceil_{\text{List}(X)}) \end{array}$$

There are other, more popular universal data types in computer science – such as natural numbers and bit-strings. A distinctive advantage of binary trees is that they natively support pairing as a primitive operation, making the encoding of complicated data structures easier.

We can define the depth of a binary tree recursively in an obvious way and introduce sets of trees T_N of depth no more than $N \in \mathbb{N}$:

$$\begin{array}{lcl} \mathit{depth} & : & T \to \mathbb{N} \\ \mathit{depth}(\texttt{nil}) & = & 0 \\ \mathit{depth}(t_1 \mathbin{.} t_2) & = & 1 + \mathit{max}(\mathit{depth}(t_1), \mathit{depth}(t_2)) \\ T_N & = & \{t \in T \mid \mathit{depth}(t) \leq N\} \end{array}$$

The programs we consider are expressions (Fig. 2) built of:

– operations for constructing (`nil`, `cons`) and destructing (`hd`, `tl`) trees;
– conditional operation (`ifnil`);
– identity function (`I`) and function composition ($\circ$).

As these programs are just expressions, we shall use both terms interchangeably. Note that there are no variables in our language. This is not an important limitation, as the combination of built-in pairing and function composition permits us to encode an arbitrary set of variables [2, 7, 8]. The semantics of our language is defined in Fig. 3. To capture the possibility of errors during program execution, we use a domain extended with a new distinct element $\perp$: $T_\perp := T \cup \{\perp\}$. In the defining equations we use wild-cards ("_") to match arbitrary items not matched in any previous equation.

$$E ::= \texttt{I} \mid \texttt{hd} \mid \texttt{tl} \mid \texttt{nil} \mid \texttt{cons}(E, E) \mid E \circ E \mid \texttt{ifnil}(E, E, E)$$

Fig. 2: Program syntax

Clearly, the class of programs we consider is far from Turing-complete, lacking any means for expressing iteration or recursion. As we are interested in decidable (extensional) equivalence, however, it is essential to consider restricted languages, as for more expressive languages equivalence is typically undecidable. This undecidability holds not only for Turing-complete languages, but even for relatively small subsets of the primitive-recursive functions, for example the class

$$
\begin{array}{ll}
[\![\bullet]\!] & : E \to T_\perp \to T_\perp \\
[\![\mathtt{I}]\!](x) & = x \\
[\![\mathtt{hd}]\!](t_1 \,.\, t_2) & = t_1 \\
[\![\mathtt{tl}]\!](t_1 \,.\, t_2) & = t_2 \\
[\![\mathtt{nil}]\!](x) & = \mathtt{nil} \\
[\![\mathtt{cons}(e_1, e_2)]\!](x) & = ([\![e_1]\!](x) \,.\, [\![e_2]\!](x)) \\
[\![e_1 \circ e_2]\!](x) & = [\![e_1]\!]([\![e_2]\!](x)) \\
[\![\mathtt{ifnil}(e_1, e_2, e_3)]\!](x) & = [\![e_2]\!](x) \text{ , if } [\![e_1]\!](x) = \mathtt{nil} \\
[\![\mathtt{ifnil}(e_1, e_2, e_3)]\!](x) & = [\![e_3]\!](x) \text{ , if } [\![e_1]\!](x) = (t_1 \,.\, t_2) \\
[\![\_]\!](\_) & = \perp
\end{array}
$$

Fig. 3: Program semantics

of elementary functions. Our class of simple programs is inspired by the simple programs introduced by Tsichritzis [14]. An important difference is that our domain consists of binary trees (and thus has a pairing operation), while Tsichritzis uses natural numbers as a domain and pairing is not definable.

3 Program Normal Forms

We can apply a number of simplifying transformations on expressions in the class we consider. Let $sel \in Sel := \{\mathtt{hd}, \mathtt{tl}\}$; the transformations we use are:

$$\mathtt{I} \circ e = e \circ \mathtt{I} = e \tag{1}$$

$$sel \circ \mathtt{cons}(e_1, e_2) = e_i \tag{2}$$

$$\mathtt{nil} \circ e = \mathtt{nil} \tag{3}$$

$$\mathtt{cons}(e_1, e_2) \circ e_3 = \mathtt{cons}(e_1 \circ e_3, e_2 \circ e_3) \tag{4}$$

$$e \circ \mathtt{ifnil}(e_1, e_2, e_3) = \mathtt{ifnil}(e_1, e \circ e_2, e \circ e_3) \tag{5}$$

$$\mathtt{ifnil}(e_1, e_2, e_3) \circ e = \mathtt{ifnil}(e_1 \circ e, e_2 \circ e, e_3 \circ e) \tag{6}$$

$$\mathtt{ifnil}(\mathtt{nil}, e_1, e_2) = e_1 \tag{7}$$

$$\mathtt{ifnil}(\mathtt{cons}(e_h, e_t), e_1, e_2) = e_2 \tag{8}$$

$$\mathtt{ifnil}(\mathtt{ifnil}(e_1, e_2, e_3), e_2', e_3') = \mathtt{ifnil}(e_1, \mathtt{ifnil}(e_2, e_2', e_3'), \mathtt{ifnil}(e_3, e_2', e_3')) \tag{9}$$

Transformations 1-6 permit to simplify instances of function composition (by either eliminating it completely or by pushing it inside subexpressions). Table 2 shows that these rules cover all cases of function composition, except for $sel_i \circ sel_j$. After these rules are applied to the condition of an if-expression, the remaining rules 7-9 allow to simplify it further.

If we exhaustively apply these transformations in a bottom-up manner, the resulting programs will be of the form shown in Fig. 4 (with an empty list of selectors being equivalent to I). We omit a detailed description of the algorithm $nf : E \to E^{nf}$ for producing normal forms, and the proofs of its properties

Table 2. Simplification rules for function composition

$\circ$	`I`	`hd/tl`	`nil`	`cons`$(\cdot,\cdot)$	`ifnil`$(\cdot,\cdot,\cdot)$
`I`	(1)	(1)	(1)	(1)	(1)
`hd/tl`	(1)	–	$\bot$	(2)	(5)
`nil`	(1)	(3)	(3)	(3)	(3)
`cons`$(\cdot,\cdot)$	(1)	(4)	(4)	(4)	(4)
`ifnil`$(\cdot,\cdot,\cdot)$	(1)	(6)	(6)	(6)	(5)

$$
\begin{array}{rcll}
E^{nf} & ::= & \texttt{nil} \mid \texttt{cons}(E^{nf}, E^{nf}) & \\
& & \mid sel_1 \circ \ldots \circ sel_n & (n \geq 0) \\
& & \mid \texttt{ifnil}(sel_1 \circ \ldots \circ sel_n, E^{nf}, E^{nf}) & (n \geq 0)
\end{array}
$$

Fig. 4: Syntax of program normal forms

(shape of normal forms, semantics preservation), as both the algorithm and the proofs appear in previous works by the author [8, 10].

We can illustrate the transformation of programs into normal form with a simple example – the composition of 2 Boolean negations. The result is, as expected, a program converting an arbitrary input tree into (an encoding of) a Boolean value, without negation.

$$
\begin{aligned}
&nf(\texttt{ifnil}(\texttt{I}, \texttt{cons}(\texttt{nil}, \texttt{nil}), \texttt{nil}) \circ (\texttt{ifnil}(\texttt{I}, \texttt{cons}(\texttt{nil}, \texttt{nil}), \texttt{nil}))) \\
&\quad = \texttt{ifnil}(\texttt{I}, \texttt{nil}, \texttt{cons}(\texttt{nil}, \texttt{nil}))
\end{aligned}
$$

The transformation rules described above are very similar to those used in supercompilation [13, 15] and deforestation [16]. In fact, we can consider the method for producing normal forms as a simple kind of supercompilation. As our language does not have loops or recursion, we do not need many of the complications involved in supercompilers for more powerful languages, such as folding, whistle, generalization.

4 Finite Adequate Test Sets for Simple Programs

4.1 Some Notions Related to Program Testing

We summarize here some definitions related to program testing used in the rest of the paper. We borrow most definitions from Budd et al. [3], but with slight differences in notation. In this subsection we consider an arbitrary set of programs P over a set of data D. The semantics of programs is given by an evaluation function $[\![\bullet]\!] : P \to D \to D$.

– Given a program $p \in P$, a *program neighborhood*[1] is any subset of programs $\Phi(p) \subseteq P$, such that $p \in \Phi(p)$.

[1] not to be confused with neighborhood analysis as a metacomputation technique

- A *test set* is a subset of data $T \subset D$ (usually tacitly assumed finite).
- A test set T is *adequate* for a program p (*relative* to a neighborhood $\Phi(p)$) if for any program $q \in \Phi(p)$ it holds:

$$(\forall d \in D, [\![p]\!](d) = [\![q]\!](d)) \leftrightarrow (\forall d \in T, [\![p]\!](d) = [\![q]\!](d))$$

The left-to-right direction in the last definition is trivial, as $T \subset D$; it is the right-to-left direction, which is important.

Relatively adequate test sets are often non-computable [3]:

- if $\Phi(p)$ are all programs in any Turing-complete language
- ... or all primitive recursive programs on $\mathbb{N}$
- ... or even all programs computing polynomials with integer coefficients
- ...

So, it is interesting to study classes of program neighborhoods, for which such tests are computable.

4.2 Subclasses of Simple Programs as Program Neighborhoods

Given some $N \in \mathbb{N}$ we define a subclass E_N^{nf} of expressions in normal form as those satisfying the following grammar:

$$\begin{array}{rcll} E_N^{nf} & ::= & \texttt{nil} \mid \texttt{cons}(E_N^{nf}, E_N^{nf}) & \\ & & \mid sel_1 \circ \ldots \circ sel_n & (0 \le n \le N) \\ & & \mid \texttt{ifnil}(sel_1 \circ \ldots \circ sel_n, E_N^{nf}, E_N^{nf}) & (0 \le n < N) \end{array}$$

It is immediately obvious from this definition that these subclasses form a cumulative hierarchy covering the whole set of normal forms E^{nf}:

- $E_N^{nf} \subsetneq E_{N+1}^{nf}$;
- $\bigcup_{N \in \mathbb{N}} E_N^{nf} = E^{nf}$.

Note also that each subclass contains infinitely many programs. By extension, we classify any program $e \in E$ to be in subclass E_N^{nf} if $nf(e) \in E_N^{nf}$. The main intuition behind the introduction of these subclasses is that programs in E_N^{nf} can only "see" at depth not more than N inside the input tree. This intuition is made formal by the following statement, which is the main result of this article:

Theorem 1. *(NTrm_fixed_MaxSelCmpLen_testable*[2]*)* $\forall N \in \mathbb{N}, \forall e_1, e_2 \in E_N^{nf}$, $(\forall t \in T_{N+1}, [\![e_1]\!](t) = [\![e_2]\!](t)) \rightarrow \forall t \in T, [\![e_1]\!](t) = [\![e_2]\!](t)$

If we compare this result with the definitions from the previous subsection, we can see that the classes E_N^{nf} can serve perfectly as program neighborhoods: if $e \in E_N^{nf}$, and we set $\Phi(e) := E_N^{nf}$, Theorem 1 shows there is a computable adequate test set for e. We devote the following subsections to an overview of the proof of this theorem.

[2] The results of this article have been formally verified in Coq. In parentheses we give the corresponding names of the theorems/lemmas in the Coq sources – `https://github.com/dkrustev/SimpleTreeExprTests`

4.3 Tree Decomposition

In order to formalize the intuition about programs in E_N^{nf} "seeing" at depth at most N inside the input tree, we consider the decomposition of a tree t into (Fig. 5):

- a tree $t_1 \in T_N$, which is isomorphic to t up to depth N;
- trees $t_2, \ldots, t_n$ corresponding to all subtrees (if any) of t with roots at depth N.

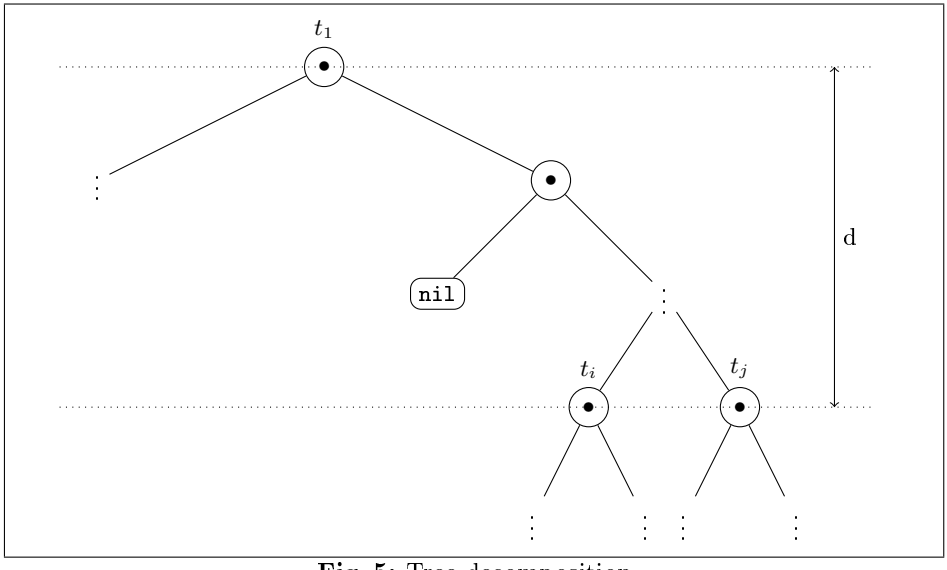

Fig. 5: Tree decomposition

To be able to recover the original tree t from its decomposition $t_1, t_2, \ldots, t_n$, we need to indicate the position of each t_i ($i \in \{2, \ldots, n\}$) inside t_1. One way to achieve it is to introduce trees with variables: given a (finite) set X of variables, the set T_X of trees with variables in X is given by the following grammar:

$$T_X \ ::= \ \texttt{nil} \mid (T_X \ . \ T_X) \mid x \qquad (x \in X)$$

The decomposition function is then $cutAt \ : \ \mathbb{N} \times T \ \to \ (X \ \to \ T) \times T_X$, $cutAt(d, t) = (\sigma, t_x)$, where:

- t_x is the tree t with all nodes at depth d replaced by variables from X
- σ is a substitution assigning the corresponding subtree to each of these variables

Example:

$$\begin{aligned} &cutAt(1, ((\texttt{nil} \ . \ \texttt{nil}) \ . \ ((\texttt{nil} \ . \ \texttt{nil}) \ . \ \texttt{nil}))) \\ &\quad = (\{x \mapsto (\texttt{nil} \ . \ \texttt{nil}), y \mapsto ((\texttt{nil} \ . \ \texttt{nil}) \ . \ \texttt{nil})\}, \\ &\qquad (x \ . \ y)) \end{aligned}$$

The action of variable substitutions is lifted to trees in the obvious way:

$$\begin{aligned} \texttt{nil}\sigma &= \texttt{nil} \\ (t_1 \ . \ t_2)\sigma &= (t_1\sigma \ . \ t_2\sigma) \\ x\sigma &= \sigma(x) \end{aligned}$$

The correctness of the decomposition function follows from the next lemma.

Lemma 1. *(vCutAt_mvSubst)* $\forall d \in \mathbb{N}, \forall t \in T, \forall \sigma, \forall t_x,\ cutAt(d,t) = (\sigma, t_x) \rightarrow t_x\sigma = t.$

As *cutAt* is defined by structural recursion over the input tree t, the proof of its correctness is by straightforward induction on t.

4.4 Existence of Adequate Test Sets

The proof of our main result relies on a couple of key observations. The first is that we can commute evaluation and substitution, provided the input tree with variables contains no variables at depth N or less. Before we write down this lemma, let us introduce some definitions. We can extend the evaluation function to work on trees with variables as well, denoted $[\![e]\!]_X \ : \ E \rightarrow T_{X_\perp} \rightarrow T_{X_\perp}$. We can use exactly the same definition as in Fig. 3, as we want the evaluation to return an error ($\perp$) whenever it encounters a variable as (top-level) input. The definition of minimum variable depth is equally straightforward:

$$\begin{aligned} minVarDepth(\texttt{nil}) &= \infty \\ minVarDepth(t_1 \ . \ t_2) &= 1 + min(minVarDepth(t_1), minVarDepth(t_2)) \\ minVarDepth(x) &= 0 \end{aligned}$$

Now our conditional commutativity property looks as follows:

Lemma 2. *(ntmvEval_ntEval)* $\forall N \in \mathbb{N}, \forall e \in E_N^{nf}, \forall X, \forall \sigma : X \rightarrow T, \forall t_x \in T_X,$ $N \leq minVarDepth(t_x) \rightarrow [\![e]\!](t_x\sigma) = ([\![e]\!]_X(t_x))\sigma.$

The proof is by induction on the structure of e. We use the condition $N \leq$ *minVarDepth* (t_x) in several cases to derive a contradiction.

The second key observation is that if we have a pair of syntactically different trees with variables, we can always build a "shallow" substitution, which – when applied to each of the 2 trees with variables – produces different ordinary trees. The substitution in question is shallow in the sense that it maps all variables to trees of depth 0 or 1.

Lemma 3. *(mvSubst_discrim)* $\forall X, \forall t_1, t_2 \in T_X, t_1 \neq t_2 \rightarrow \exists \sigma,\ (\forall x \in X,\ \sigma(x)$ $\in T_1) \ \wedge t_1\sigma \neq t_2\sigma.$

Proof sketch: there must be at least one pair of corresponding subtrees t_1' and t_2' with different root nodes

- if neither root is a variable, then the trivial substitution will do: $\sigma(x) =$ $\texttt{nil}, \forall x \in X$;

- if only one root is a variable, say y
  - if the other root is `nil`, then $\sigma(x) =$ if $x = y$ then (`nil` . `nil`) else `nil`;
  - if the other root is $(t_3 \,.\, t_4)$, then $\sigma(x) =$ if $x = y$ then `nil` else (`nil` . `nil`);
- if $t'_1 = x, t'_2 = y$, then we can use:

$$\begin{aligned} &\sigma(x) = \texttt{nil} \\ &\sigma(y) = (\texttt{nil} \,.\, \texttt{nil}) \\ &\sigma(z) = \texttt{nil} \qquad\qquad \forall z \in X, z \neq x, z \neq y. \square \end{aligned}$$

Armed with these observations, we can proceed with establishing the existence of finite adequate test sets:

Lemma 4. *(NTrm_fixed_MaxSelCmpLen_testable_aux)* $\forall N, \forall e_1, e_2 \in E_N^{nf}$, $(\exists t \in T, [\![e_1]\!](t) \neq [\![e_2]\!](t)) \rightarrow \exists t \in T_{N+1}, [\![e_1]\!](t) \neq [\![e_2]\!](t)$.

Proof sketch:

- let $t \in T$, s.t. $[\![e_1]\!](t) \neq [\![e_2]\!](t)$
- let $(\sigma, t_x) = cutAt(N, t)$
- then $[\![e_1]\!](t_x\sigma) \neq [\![e_2]\!](t_x\sigma)$ (by Lemma 1)
- commute evaluation and substitution: $([\![e_1]\!]_X(t_x))\ \sigma \neq ([\![e_2]\!]_X(t_x))\ \sigma$ (by Lemma 2)
  - possible because $cutAt(N, t) = (\sigma, t_x)$ ensures the required condition $N \leq minVarDepth(t_x)$
- so $[\![e_1]\!]_X(t_x) \neq [\![e_2]\!]_X(t_x)$
- the most interesting case is when both evaluation results are $\neq \bot$
- then (by Lemma 3) we can find σ' s.t. all $\sigma'(x) \in T_1$ and $([\![e_1]\!]_X(t_x))\sigma' \neq ([\![e_2]\!]_X(t_x))\sigma'$
- commute substitution and evaluation (again by Lemma 2): $([\![e_1]\!](t_x\sigma')) \neq ([\![e_2]\!](t_x\sigma'))$
- let $t' = t_x\sigma'$; we have $t' \in T_{N+1}$ and $[\![e_1]\!](t') \neq [\![e_2]\!](t')$. $\square$

Now it suffices to remark that Lemma 4 is just the contrapositive of Theorem 1, which concludes the proof of our main result.

4.5 Test Set Optimality

If we consider the whole set E_N^{nf} as a neighborhood of the program $e \in E_N^{nf}$, we cannot substantially improve the size of the adequate test set provided by Theorem 1, as the following theorem shows:

Theorem 2. *(undiscrTerms_exist)* $\forall N \in \mathbb{N}, \exists e_1, e_2 \in E_{N+1}^{nf}$, *such that* $\forall t \in T_{N+1}$, $[\![e_1]\!](t) = [\![e_2]\!](t)$ *and* $\exists t \in T, [\![e_1]\!](t) \neq [\![e_2]\!](t)$.

Proof sketch: it suffices to take

$$\begin{aligned} e_1 &= \texttt{hd} \circ e \\ e_2 &= \texttt{tl} \circ e, \qquad \text{where:} \\ e \ &= \underbrace{\texttt{hd} \circ \ldots \circ \texttt{hd}}_{N \text{ times}} \end{aligned}$$

$\square$

5 Decidability of Equivalence of Simple Programs

One direct application of the existence of finite adequate test sets is the decidability of equivalence for our class of simple programs. If we consider 2 programs $e_1, e_2 \in E$, we can proceed as follows:

- find the smallest N, such that $nf(e_1), nf(e_2) \in E_N^{nf}$;
- test if $[\![e_1]\!](t) = [\![e_2]\!](t)$ for all $t \in T_{N+1}$

If there is some t, for which the 2 programs return different results, they are clearly not equivalent. If, however, there is no such $t \in T_{N+1}$, then by Theorem 1 it follows that the 2 programs are equivalent.

The asymptotic complexity of this decision procedure is superexponential. Indeed, the number of unlabeled binary trees of depth no more than N is given by the following recurrence:

$$a_0 = 1$$
$$a_{N+1} = a_N^2 + 1$$

According to OEIS [12], $a_N \asymp c^{2^{N+1}}$ where $c = 1.2259\ldots$, which directly gives an superexponential bound for our algorithm. We leave as future work the search for algorithms of lower complexity. One idea that might work is to consider smaller subclasses of expressions of the following form:

$$\begin{array}{rcll} E_S^{nf} & ::= & \texttt{nil} \mid \texttt{cons}(E_S^{nf}, E_S^{nf}) & \\ & & \mid sel_1 \circ \ldots \circ sel_n & (sel_1 \circ \ldots \circ sel_n \in S) \\ & & \mid \texttt{ifnil}(sel_1 \circ \ldots \circ sel_n, E_S^{nf}, E_S^{nf}) & (sel_1 \circ \ldots \circ sel_n \in S) \end{array}$$

If we can adapt our proof of existence of adequate test sets to this kind of smaller program neighborhoods, we can hope to get asymptotically smaller tests sets and as a result – a faster decision procedure for program equivalence.

6 Related Work

As already noted, the main results in this paper are very similar to – and to some extent inspired by – the work of Tsichritzis [14]. Tsichritzis starts with a subclass of primitive-recursive programs on natural numbers – namely, those having no nested loops – and first shows that all such programs can be represented as compositions of several simple operations. Then finite test sets are defined for this language, and – as a consequence – a decision procedure for program equivalence. We, on the other hand, start directly with a language consisting of expressions, which are composed of several simple operations on binary trees. Still, the languages considered are very similar in spirit, modulo differences in the data domains. The use of binary trees is an important advantage of our approach: as we have already noted, binary trees come with pairing as a built-in primitive, and it permits easy encoding of arbitrary data structures. The language, treated by Tsichritzis, is too weak to encode arbitrary pairing.

Binary trees have often been used in practical programming languages since the early days of Lisp. While most theoretical models of computation use either natural numbers or sequences over a fixed alphabet as the only data structure, several authors have noted the usefulness of binary trees in a more theoretical setting as well, for example Jones [7]. The variable-free nature of our language has its roots in Backus' FP [2], but Jones has proposed a similar language of a single variable [7].

The main results we report are enabled by the specific shapes of normal forms that we can produce by supercompilation-like program transformations. In this respect, the current work is an offshoot of some of the author's previous research on supercompilation [8,10]. Supercompilation – and metacomputation techniques in general – are the basis of several other methods for test generation [1, 9, 10]. Abramov's neighborhood testing [1] ensures strong adequacy properties for the generated test sets and covers arbitrary languages, but it is not guaranteed to terminate. Our skeleton testing method [10] produces adequate test sets for a Turing-complete functional language, but the program neighborhoods are finite and the test sets actually use program expressions instead of simple data values. We have also proposed a test-generation method based on metacomputation (used for program inversion) [9], which is more practically oriented, but without any formal adequacy guarantees.

The literature on program testing techniques is too big to review here. We just note several methods, which – similarly to ours – produce finite adequate test sets[3] for restricted classes of programs:

- the already discussed work of Tsichritzis [14] – primitive recursive programs on natural numbers without nested loops;
- programs computing multivariate polynomials of a known degree with integer coefficients [4, 6];
- programs computing multivariate polynomials of unknown degree with natural coefficients [5, 11].

7 Conclusions and Future Work

We have presented a class of simple programs operating on binary trees, which can be split into subclasses, such that for each subclass there exists a finite adequate test set. As a direct consequence, equivalence of programs in the whole class is decidable. The definition of the subclasses is made possible by converting programs into a normal form with specific shape, through supercompilation-like transformations.

The practical application of the equivalence decision procedure is hindered by its superexponential complexity. We have already mentioned some ideas that might help reduce this complexity, but more work is needed to flesh them out. The size of the test set (which is the cause for the decision procedure complexity) is also a hurdle for the practical application of the test generation method. A

[3] Possibly for a slightly different definition of "adequate"

potential approach for reducing the test set size, which we plan to explore, is to use trees with variables as test inputs and outputs. Another problem with applying the method for practical program testing is the use of a very restricted language, which does not by itself permit writing many interesting programs. One possibility is to study the use of the language of simple programs discussed here as the core of a Turing-complete language. For example, we can use a simple imperative language with while loops, similar to the ones used by Jones [7] and in our earlier work [8], with the language of simple programs being embedded as a sublanguage of expressions. The key research problem will be how to extend the test generation method from expressions to programs in the full language.

Another interesting problem, which we may consider in the future, is the more precise characterization of the expressiveness of the class of simple programs we have studied.

Acknowledgments I would like to thank Alexei Lisitsa for comments that helped improve the presentation of this article.

References

1. Abramov, S.M.: Metacomputation and program testing. In: Proceedings of the 1st International Workshop on Automated and Algorithmic Debugging. pp. 121–135. Linköping University, Linköping, Sweden (1993)
2. Backus, J.: Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Communications of the ACM 21(8), 613–641 (Aug 1978)
3. Budd, T.A., Angluin, D.: Two notions of correctness and their relation to testing. Acta Informatica 18(1), 31–45 (Nov 1982)
4. DeMillo, R.A., Lipton, R.J.: A probabilistic remark on algebraic program testing. Information Processing Letters 7(4), 193–195 (1978)
5. Govindarajan, K.: A note on polynomials with non-negative integral coefficients. Tech. rep., Department of Computer Science, State University of New York at Buffalo (1995)
6. Howden, W.E.: Algebraic program testing. Acta Informatica 10(1), 53–66 (1978), http://dx.doi.org/10.1007/BF00260923
7. Jones, N.D.: Computability and Complexity from a Programming Perspective. Foundations of Computing, MIT Press, Boston, London, 1 edn. (1997)
8. Krustev, D.: A simple supercompiler formally verified in Coq. In: Nemytykh, A.P. (ed.) Proceedings of the Second International Workshop on Metacomputation in Russia (META 2010). pp. 102–127 (2010)
9. Krustev, D.: A metacomputation toolkit for a subset of F# and its application to software testing. In: Klimov, A.V., Romamenko, S.A. (eds.) Third International Valentin Turchin Workshop on Metacomputation. pp. 165–183 (2012)
10. Krustev, D.N.: Software test generation using program skeletons. PhD thesis Technical Report PRE 23/01, Wroclaw Polytechnic, Wroclaw, Poland (2001)
11. Maolepszy, J.: Reconstruction of extended polynomials from the finite number of examples. Tech. rep., Jagiellonian University, Institute Of Computer Science, Krakow (1996)

12. OEIS: sequence A003095. https://oeis.org/A003095, [Online; accessed 2016-05-08]
13. Sørensen, M.H., Glück, R.: Introduction to supercompilation. In: Hatcliff, J., Mogensen, T., Thiemann, P. (eds.) Partial Evaluation: Practice and Theory. Lecture Notes in Computer Science, vol. 1706, pp. 246–270. Springer-Verlag (1999)
14. Tsichritzis, D.: The equivalence problem of simple programs. J. ACM 17(4), 729–738 (1970)
15. Turchin, V.: The concept of a supercompiler. ACM Transactions on Programming Languages and Systems 8(3), 292–325 (July 1986)
16. Wadler, P.: Deforestation: Transforming programs to eliminate trees. Theor. Comput. Sci. 73(2), 231–248 (1990)