

Time-complexity semantics for feasible affine recursions (extended abstract)

NORMAN DANNER AND JAMES S. ROYER

ABSTRACT. The authors' ATR programming formalism is a version of call-by-value PCF under a complexity-theoretically motivated type system. ATR programs characterize the type-level ≤ 2 basic feasible functions (ATR-types are confined to levels 0, 1, and 2). A limitation of the original version of ATR is that the only directly expressible recursions are tail-recursions. Here we extend ATR so that a broad range of affine recursions are directly expressible. In particular, the revised ATR can fairly naturally express the classic insertion- and selection-sort algorithms, thus overcoming a sticking point of most prior implicit-complexity-based formalisms. The paper's main work is in extending and simplifying the original time-complexity semantics for ATR to develop a set of tools for extracting and solving the higher-type recurrences arising from feasible affine recursions.

1. TWO ALGORITHMS IN SEARCH OF A TYPE-SYSTEM

As Hofmann [9] has noted, a problem with implicit characterizations of complexity classes is that they often fail to capture many natural *algorithms*—usually because the complexity-theoretic types used to control primitive recursion impose draconian restrictions on programming. Here is an example. In Bellantoni and Cook's [3] and Leivant's [11] well-known characterizations of the polynomial-time computable functions, a recursively-computed value is prohibited from driving another recursion. But, for instance, the recursion clause of insertion-sort has the form $\text{ins_sort}(\text{cons}(a, l)) = \text{insert}(a, \text{ins_sort}(l))$, where insert is defined by recursion on its second argument; selection-sort presents analogous problems.

Hofmann [9, 8] addresses this problem by noting that the output of a non-size-increasing program (such as ins_sort) should be permitted to drive another recursion, as it cannot cause the sort of complexity blow-up the B-C-L restrictions guard against. To incorporate such recursions, Hofmann defines a higher-order language with typical first-order types and a special type \diamond through which functions defined recursively must “pay” for any use of size-increasing constructors, in effect guaranteeing that there is no size increase. Through this scheme Hofmann is able to implement many natural algorithms while still ensuring that any typable program is non-size-increasing polynomial-time computable (Aehlig and Schwichtenberg [1] sketch an extension that captures all of polynomial-time).

Our earlier paper [5, 6], hereafter referred to as *ATS*, takes a different approach to constructing a usable programming language with guaranteed resource usage. We introduce a type-2 programming formalism called ATR (for Affine Tail Recursion, which we rechristen in this paper as Affine *Tiered* Recursion) based on PCF. ATR's type system is motivated by the tiering and safe/normal notions of [11] and [3] and serves to control the size of objects. Instead of restricting to primitive recursion, ATR has an operator for recursive definitions; affine types and explicit clocking on the operator serve to control time. We give a denotational semantics to ATR types and terms in which the size restrictions play a key part. This allows us, for example, to give an ATR *definition* of a primitive-recursion-on-notation combinator (with appropriate types and without explicit bounding terms) that preserves feasibility. We also give a *time-complexity semantics* and use it to prove that the type-2 functions definable in ATR are exactly the type-2 basic feasible functionals (an extension of

polynomial-time computability to type-2) of Mehlhorn [12] and Cook and Urquhart [4].¹ Moreover, our underlying model of computation (and complexity) is just a standard abstract machine that implements call-by-value PCF. However, ATR is still somewhat limited as its only base type is binary words and the only recursions allowed are tail-recursions.

What is new in this paper. In this paper we extend ATR to encompass a broad class of feasible affine recursions. We demonstrate these extensions by giving fairly direct and natural versions of insertion- and selection-sorts on lists (the latter in the appendix). As additional evidence of ATR's support for programming we do not add lists as a base type, but instead show how to implement them over ATR's base type of binary words.

The technical core of this paper is a simplification and generalization of the time-complexity semantics of *ATS*. We construct a straightforward framework in which recursion schemes in ATR lead to time-complexity recurrences that must be solved to show that these schemes preserve feasibility. This gives a route to follow when adding new forms of recursion to ATR. We follow this route to show that the recursions used to implement lists and insertion-sort are (second-order) polynomial-time bounded. We also discuss how to extend these results to handle the recursions present in selection-sort. Thus along with significantly extending our existing system to the point where many standard algorithms can be naturally expressed, we also provide a set of basic tools for further extensions.

2. PROGRAMMING IN ATR

The ATR formalism. We briefly summarize the ATR formalism here; for details see Appendix A. An ATR base type has the form N_L , where *labels* L are elements of the set $(\Box\Diamond)^* \cup \Diamond(\Box\Diamond)^*$ (our use of \Diamond is not directly related to Hofmann's). Roughly, we can think of type- N_ϵ values as basic string inputs, type- N_\Diamond values as the result of polynomial-time computations over N_ϵ -values, type- $N_{\Box\Diamond}$ -values as the result applying an oracle (a type-1 input) to N_\Diamond -values, type- $N_{\Diamond\Box}$ values as the result of polynomial-time computations over $N_{\Box\Diamond}$ -values, etc. N_L is called an *oracular* (respectively, *computational*) type when $L \in (\Box\Diamond)^*$ (respectively, $\Diamond(\Box\Diamond)^*$). We let \mathbf{b} (possibly decorated) range over base types. Function types are formed as usual from the base types. The term forming operations correspond to adding and deleting a left-most bit (c_0 , c_1 , and d), testing whether a word begins with a 0 or a 1 (t_0 and t_1), and a conditional. We include an operator **down**; the intended interpretation of **down** s t is s if $|s| \leq |t|$ and ϵ otherwise. The recursion operator is **crec**, standing for clocked recursion.

Type contexts are split (after Barber and Plotkin's DILL [2]) into intuitionistic and affine zones. Variables in the former correspond to the usual \rightarrow introduction and elimination rules; variables in the latter are intended to be recursively defined, and the **crec**-Introduction rule

$$(*) \quad \frac{\Gamma; _ \vdash K : N_\Diamond \quad \Gamma, \vec{v} : \vec{\mathbf{b}}; f : \vec{\mathbf{b}} \rightarrow \mathbf{b} \vdash t : \mathbf{b}}{\Gamma; _ \vdash \text{crec } K(\lambda_r f. \lambda \vec{v}. t) : \vec{\mathbf{b}} \rightarrow \mathbf{b}}$$

serves as both introduction and elimination rule for the implicit \multimap types (here $\vec{\mathbf{b}} = \mathbf{b}_1, \dots, \mathbf{b}_k$ and $\vec{v} : \vec{\mathbf{b}}$ stands for $v_1 : \mathbf{b}_1, \dots, v_k : \mathbf{b}_k$). The side-conditions on $(*)$ are that f occurs in cons-tail position² in t and if $\mathbf{b}_i \leq \mathbf{b}_1$ ($\mathbf{b} \leq \mathbf{b}_1$) then \mathbf{b}_i (\mathbf{b}) is oracular. The typing rules enforce a “one-use” restriction

¹ These kinds of results may also have applications in the type of static analysis for time-complexity that Fredriksen and Jones [7] investigate.

² Informally, f occurs in *cons-tail position* in t if in the parse-tree of t a path from the root to a complete application of f passes through only conditional branches (not tests), c_0 , c_1 , and the left-argument of **down**; $\text{tail_len}(f, t)$ is defined to be the maximum number of c_a operations not below any **down** node in any such path.

on affine variables by disallowing their occurrence as a free variable in both arguments of **down**, the argument of an application, the test of a conditional, or anywhere in a **crec**-term.

Motivated by the approach of Jones [10], we define the cost of evaluation to be the size of a call-by-value evaluation derivation. This is essentially equivalent to the abstract machine-based cost model of *ATS*, but the derivation-based model helps avoid considerable bookkeeping clutter. Values are string constants, oracles, or abstractions. Environments map term variables to values or to closures over **crec** terms. A closure $t\rho$ consists of a term and an environment. The evaluation relation has the form $t\rho \downarrow z\theta$ where $t\rho$ is a closure and z is a value. The derivation rules for the evaluation are mostly straightforward and mimic the action of the abstract machine of *ATS*; for example, we have

$$\frac{\rho(x) \downarrow z\theta}{x\rho \downarrow z\theta} \quad \frac{t\rho \downarrow (az)\theta}{(d\ t)\rho \downarrow z\theta} \quad \frac{s\rho \downarrow w\zeta \quad t\rho \downarrow z\theta \quad |w| \leq |z|}{(\text{down } st)\rho \downarrow w\zeta}$$

The evaluation rule for **crec** terms is

$$\overline{(\text{crec } a(\lambda_r f. \lambda \vec{v}. t))\rho \downarrow (\lambda \vec{v}. \text{if } |a| < |v_1| \text{ then } t \text{ else } \varepsilon)\rho[f \mapsto \text{crec}(0a)(\lambda_r f. \lambda \vec{v}. t)]}$$

which shows how unwinding the recursion increments the clock by one step. The cost of most inference rules is 1, except the **down** inference rules have cost $2|z| + 1$ and environment and oracle evaluation have length cost (so, e.g., the cost of the environment rule shown above is $\max(|z|, 1)$ when z is of base type, 1 otherwise).

Implementing lists and sorting. We implement lists of binary words via concatenated self-delimiting strings. Specifically, we code the word $w = b_0 \dots b_{k-1}$ as $s(w) = 1b_01b_1 \dots 1b_{k-1}0$ and the list $\langle w_0, \dots, w_{k-1} \rangle$ as $s(w_0) \oplus \dots \oplus s(w_{k-1})$, where \oplus is the concatenation operation. We show the **head** operation in Figure 1 and give all the code in Appendix B.³ Note that the **cons**, **head**, and **tail** programs all use **cons-tail** recursion. Insertion-sort is expressed in essentially its standard form, as in Figure 1. This implementation requires another form of recursion, in which the complete application of the recursively-defined function appears in an argument to some operator. In the later part of Section 4 we show how this *recursion in an argument* can be incorporated into ATR. We implement selection-sort in Appendix B. Selection-sort requires yet another form of recursion (a generalization of **cons-tail** recursion); we discuss how to incorporate it into ATR in Section 4.

Our **head** and **ins_sort** programs use the **down** operator to coerce the type N_\diamond to N_ε . Roughly, **down** is used in places where our type-system is not clever enough to prove that the result of a recursion is of size no larger than one of the recursion’s initial arguments; the burden of supplying these proofs is shifted off to the correctness argument for the recursion. A cleverer type system (say, along the lines of Hofmann’s [8]) could obviate many of these **down**’s, but at the price of more complex syntax (i.e., typing), semantics (of values and of time-complexities), and, perhaps, pragmatics (i.e., programming). Our use of **down** gives us a more primitive (and intensional) system than found in pure implicit complexity,⁴ but it also gives us a less cluttered setting to work out the basics of complexity-theoretic compositional semantics—the focus of the rest of the paper. Also, in practice the proofs that the uses of **down** forces into the correctness argument are for the most part obvious, and thus not a large burden on the programmer.

³ In these code samples, `letrec f=s in t end` abbreviates `t[f ↦ crec ε(λrf.s)]` and we use the ML notation `fn x ⇒ ...` for λ-abstraction.

⁴Leivant’s *recursion under a high-tier bound* [11, §3.1] implements a similar idea.

```

val head :  $N_\varepsilon \rightarrow N_\varepsilon =$ 
  fn l  $\Rightarrow$  letrec dec :  $N_\varepsilon \rightarrow N_\diamond \rightarrow N_\diamond =$ 
    fn b x  $\Rightarrow$  if t1(x) then
      if t0(d x) then c0(dec b (d(d(x)))) else c1(dec b (d(d(x))))
    else  $\varepsilon$ 
  in down (dec l l)(l) end

```

```

val insert :  $N_\varepsilon \rightarrow N_\varepsilon \rightarrow N_\diamond =$ 
  fn w l  $\Rightarrow$  letrec ins :  $N_\varepsilon \rightarrow N_\varepsilon \rightarrow N_\diamond =$ 
    fn b l'  $\Rightarrow$  if l' then
      if leq w head(l') then cons w l'
      else cons (head l') (ins b (tail l'))
    else cons w nil
  in ins l l end

```

```

val ins_sort :  $N_\varepsilon \rightarrow N_\diamond =$ 
  fn l  $\Rightarrow$  letrec isort :  $N_\varepsilon \rightarrow N_\varepsilon \rightarrow N_\diamond =$ 
    fn b l' = if l' then insert (head l') (down (isort b (tail l')) l') else  $\varepsilon$ 
  in isort l l end

```

FIGURE 1. The list-head operation and insertion-sort in ATR.

3. SOUNDNESS THEOREMS

In this section we rework the Soundness Theorem of *ATS* to set up the framework for such theorems, and then use the framework to handle the recursions used to implement insertion-sort (we discuss selection-sort in Section 4). The key technical notion is that of *bounding* a closure $t\rho$ by a *time-complexity*, which provides upper bounds on the cost of evaluating $t\rho$ to a value $z\theta$ as well as the *potential* cost of using $z\theta$. The potential of a base-type closure is just its (denotation's) length, whereas the potential of a function f is a function that maps potentials p to the time complexity of evaluating f on arguments of potential p . The bounding relation gives a *time-complexity semantics* for ATR-terms; a *soundness theorem* asserts the existence of a bounding time-complexity for every ATR term. In this paper, our soundness theorems also assert that the bounding time-complexities are *safe*, which in particular implies type-2 polynomial size and cost bounds for the closure. We thereby capture the Soundness, polynomial-size-boundedness, and polynomial-time-boundedness theorems of *ATS* (the *value semantics* for the meaning of ATR terms and corresponding soundness theorem are unchanged).

Soundness for tail-recursion. We start by defining *cost*, *potential*, and *time-complexity* types, all of which are elements of the simple product type structure over the *time-complexity base types* $\{\mathbf{T}\} \cup \{\mathbf{T}_L \mid L \text{ is a label}\}$. The only cost type is \mathbf{T} , and for each ATR-type σ we define the potential type $\langle\langle\sigma\rangle\rangle$ and time-complexity type $\|\sigma\|$ by $\langle\langle\mathbf{N}_L\rangle\rangle = \mathbf{T}_L$, $\langle\langle\sigma \rightarrow \tau\rangle\rangle = \langle\langle\sigma\rangle\rangle \rightarrow \|\tau\|$, and $\|\tau\| = \mathbf{T} \times \langle\langle\tau\rangle\rangle$. Write *cost*(\cdot) and *pot*(\cdot) for the left- and right-projections on $\|\tau\|$. We extend $\|\cdot\|$ to type contexts by introducing variables x_c and x_p for each ATR-variable x and setting $\|\Gamma\| = \cup_{(x:\sigma) \in \Gamma} \{x_c : \mathbf{T}, x_p : \langle\langle\sigma\rangle\rangle\}$. A *time-complexity denotation* (*t.c. denotation*) of t.c. type γ w.r.t. a t.c. environment Σ is a function $X : \Sigma\text{-Env} \rightarrow \gamma$. The projections *cost* and *pot* extend to t.c. denotations in the obvious way. Because of the “looking into the future” aspect of time-complexities, the bounding relation is defined by

looking at the form of the value to which a closure evaluates, then describing the potential behavior of that value.

DEFINITION 1.

- (1) Suppose $t\rho$ is a closure and $z\theta$ a value, both of type τ ; χ a time-complexity of type $\|\tau\|$; and q a potential of type $\langle\langle\tau\rangle\rangle$. Define the *bounding relations*⁵ $t\rho \sqsubseteq^\tau \chi$ and $z\theta \sqsubseteq_{\text{pot}}^\tau q$ as follows:⁶
 - (a) $t\rho \sqsubseteq^\tau \chi$ if $\text{cost}(t\rho) \leq \text{cost}(\chi)$ and if $t\rho \downarrow z\theta$, then $z\theta \sqsubseteq_{\text{pot}}^\tau \text{pot}(\chi)$;
 - (b) $z\theta \sqsubseteq_{\text{pot}}^b q$ if $|z| \leq q$;
 - (c) $(\lambda v.t)\theta \sqsubseteq_{\text{pot}}^{\sigma \rightarrow \tau} q$ if for all values $z\eta$, if $z\eta \sqsubseteq_{\text{pot}}^\sigma p$, then $t\theta[v \mapsto z\eta] \sqsubseteq^\tau q(p)$.
 - (d) $O\theta \sqsubseteq_{\text{pot}}^{\sigma \rightarrow \tau} q$ if for all values $z\eta$, if $z\eta \sqsubseteq_{\text{pot}}^\sigma p$, then $(O(z\eta))[] \sqsubseteq^\tau q(p)$.
- (2) For $\rho \in \Gamma\text{-Env}$ and $\varrho \in \|\Gamma\|\text{-Env}$, we write $\rho \sqsubseteq \varrho$ if for all $v \in \text{Dom } \rho$ we have that $v\rho \sqsubseteq (\varrho(v_c), \varrho(v_p))$.
- (3) For an ATR-term $\Gamma; \Delta \vdash t : \tau$ and a time-complexity denotation X of type $\|\tau\|$ w.r.t. $\|\Gamma; \Delta\|$, we say $t \sqsubseteq X$ if for all $\rho \in (\Gamma; \Delta)\text{-Env}$ and $\varrho \in \|\Gamma; \Delta\|\text{-Env}$ such that $\rho \sqsubseteq \varrho$ we have that $t\rho \sqsubseteq X\varrho$.

We define second-order polynomial expressions of tally, potential, and time-complexity types as expected using the operations $+$, $*$, and \vee (binary maximum). Polynomials of computational type may be added and multiplied, whereas the maximum may be taken of two polynomials of any base type (details are in Appendix A). Of course, a polynomial $\Sigma \vdash p : \gamma$ corresponds to a t.c. denotation of type γ w.r.t. Σ in the obvious way. We shall frequently write p_p for $\text{pot}(p)$.

DEFINITION 2. Let γ be a potential type, \mathbf{b} a time-complexity base type, p a potential polynomial, and suppose $\Sigma \vdash p : \gamma$.

- (1) p is **b**-strict w.r.t. Σ when $\text{tail}(\gamma) \leq \mathbf{b}$ and every unshadowed⁷ free-variable occurrence in p has a type with $\text{tail} < \mathbf{b}$.
- (2) p is **b**-chary w.r.t. Σ when $\gamma = \mathbf{b}$ and $p = p_1 \vee \dots \vee p_m$ with $m \geq 0$ where $p_i = (vq_1 \dots q_k)$ with each q_i **b**-strict.
- (3) p is **b**-safe w.r.t. Σ if:
 - (a) γ is a base type and $p = q \odot_{\mathbf{b}} r$ where q is **b**-strict and r is **b**-chary, $\odot_{\mathbf{b}} = \vee$ if \mathbf{b} is oracular, and $\odot_{\mathbf{b}} = +$ if \mathbf{b} is computational.
 - (b) $\gamma = \sigma \rightarrow (\mathbf{T} \times \tau)$ and $\text{pot}(pv)$ is **b**-safe w.r.t. $\Sigma, v : \sigma$.
- (4) A t.c. polynomial $\Sigma \vdash q : \mathbf{T} \times \gamma$ is **b**-safe if $\text{pot}(q)$ is.
- (5) A t.c. denotation X of type γ w.r.t. Σ is **b**-safe if X is bounded by a **b**-safe t.c. polynomial $\Sigma \vdash p : \gamma$.

The Soundness Theorem of *ATS* asserts that every tail-recursive term is bounded by a t.c. denotation for which the cost component is bounded by a type-2 polynomial in the lengths of t 's free variables. In the next subsection, we extend this to cons-tail recursion and prove that the bounding t.c. denotation is in fact safe. In particular, we also have that the potential of t 's denotation is bounded by a safe polynomial. At base type, this latter statement corresponds to the “poly-max” bounds that can be computed for Bellantoni-Cook and Leivant-style tiered functions (e.g., [3, Lemma 4.1]).

⁵We call these *approximating relations* in *ATS*.

⁶We will drop the superscript when it is clear from context.

⁷Roughly, a free-variable occurrence is *unshadowed* if it is not eventually in the argument of a vacuous abstraction.

Soundness for cons-tail-recursion. For the remainder of this subsection t is a term such that f is in cons-tail position in t and for which we have a typing $\Gamma, \vec{v}:\vec{b}; f:\vec{b} \rightarrow \mathbf{b} \vdash t:\mathbf{b}$. We write $\Gamma_{\vec{v}}$ for the type context $\Gamma, \vec{v}:\vec{b}$. Define the terms $C_\ell = \text{crec}(0^\ell a)(\lambda_r f. \lambda \vec{v}. t)$ and $T_\ell = \text{if } |0^\ell a| < |v_1| \text{ then } t \text{ else } \varepsilon$, and for any environment ρ , set $\rho_\ell = \rho[f \mapsto C_\ell]$. The main difficulty in proving soundness is constructing a bounding t.c. denotation for crec terms. A key component in the construction is the Affine Decomposition Theorem Section 14 of *ATS*, which describes how to compute the time-complexity of a term in which f occurs affinely and in tail position. To state it, we need some definitions.

DEFINITION 3. Let X and Y be t.c. denotations of type $\|\sigma \rightarrow \tau\|$ and $\|\sigma\|$, respectively.

- (1) For a potential $p:\mathbf{T}_L$, $\text{val } p = (1 \vee p, p)$; if p is of higher type, then $\text{val } p = (1, p)$. For a t.c. environment ϱ and ATR variable v we write $\varrho[v \mapsto \chi]$ for $\varrho[v_c, v_p \mapsto \text{cost}(\chi), \text{pot}(\chi)]$.
- (2) If Y is w.r.t. $\|\Gamma, v:\sigma'\|$, then $\mathbb{A}_* v. Y = \mathbb{A}_\varrho(1, \mathbb{A}_{v_p}. Y(\varrho[v \mapsto \text{val } v_p]))$ is a t.c. denotation of type $\|\sigma' \rightarrow \sigma\|$ w.r.t. $\|\Gamma\|$.
- (3) $X \star Y = \mathbb{A}_\varrho(\text{cost}(X\varrho) + \text{cost}(Y\varrho) + \text{cost}(\chi) + 1, \text{pot}(\chi))$ is a t.c. denotation of type $\|\tau\|$, where $\chi = \text{pot}(X\varrho)(\text{pot}(Y\varrho))$ (we write $\mathbb{A}_\varrho \dots$ for $\varrho \mapsto \dots$).
- (4) $\text{dally}(\ell, X) = \mathbb{A}_\varrho(\ell + \text{cost}(X\varrho), \text{pot}(X\varrho))$ and for $\|\sigma\| = \mathbf{T} \times \mathbf{T}_L$, $\text{pad}(\ell, Y) = \mathbb{A}_\varrho(\text{cost}(Y\varrho), \ell + \text{pot}(Y\varrho))$.
- (5) For $\|\sigma\| = \mathbf{T} \times \mathbf{T}_L$ and Z also a t.c. denotation of type $\|\sigma\|$, $(Z \uplus Y)\varrho = (\text{cost}(Z\varrho) + \text{cost}(Y\varrho), \text{pot}(Z\varrho) \vee \text{pot}(Y\varrho))$.

THEOREM 1 (Decomposition Theorem). Suppose $t \sqsubseteq X$ and Y_i is such that if $ft_1 \dots t_k$ is a complete application of f in t , then $t_i \sqsubseteq Y_i$. Then

$$t \sqsubseteq \mathbb{A}_\varrho(X\varrho_\varepsilon \uplus \text{pad}(\text{tail_len}(f, t), \varrho f \star Y_{1\varrho_\varepsilon} \star \dots \star Y_{k\varrho_\varepsilon}))$$

where $\varrho_\varepsilon = \varrho[f \mapsto \mathbb{A}_* \vec{v}.(1, 0)]$ and $\text{tail_len}(f, t)$ is defined in Footnote 2.

Intuitively, the cost of “getting to” the recursive call is covered by $X\varrho_\varepsilon$, and the cost of the call itself by $\varrho f \star Y_{1\varrho_\varepsilon} \star \dots \star Y_{k\varrho_\varepsilon}$, taking into account any c_a operations after the call (this is an over-estimate if no recursive call is made). The potential (size in this case, since t is of base type) is either independent of any complete application of f or is equal to the size of such an application, again taking into account later c_a operations.

DEFINITION 4. A *decomposition function* for t is a function $d(\varrho^{\|\Gamma_{\vec{v}}\|-\text{Env}}, \chi^{\|\gamma\|}) : \|\mathbf{b}\|$ such that $t \sqsubseteq \mathbb{A}_\varrho.d(\varrho_\varepsilon, \varrho f)$.

Recalling the evaluation rule for crec and the definition of \sqsubseteq , we see that we must understand how the closure $T_0\rho_1$ is evaluated for appropriate ρ . It is easy to see that in such an evaluation, the only sub-evaluations of closures over terms of the form T_m are evaluations of closures of the form $T_m\rho_{m+1}[\vec{v} \mapsto \vec{z}\vec{\theta}]$ for some closures $z_i\theta_i$. For the closure $T_0\rho_1$ we say that *the clock is bounded by K* if in every such sub-evaluation we have that $|z_1| < K$.

For a decomposition function d define $\Phi_{d,K}(n) : \|\Gamma_{\vec{v}}\|-\text{Env} \rightarrow \|\mathbf{b}\|$ by

$$\Phi_{d,K}(0) = \mathbb{A}_\varrho.(2K + 1, 0)$$

$$\Phi_{d,K}(n + 1) = \mathbb{A}_\varrho.\text{dally}(2K + 1, d(\varrho_\varepsilon, \text{dally}(2, (\mathbb{A}_* \vec{v}.\Phi_{d,K}(n))\varrho)) \vee (1, 0))$$

We will use $\Phi_{d,K}$ to bound T_ℓ .

THEOREM 2 (Recomposition Lemma). Suppose d is a decomposition function for t , $\rho \in \Gamma_{\vec{v}}-\text{Env}$, $\varrho \in \|\Gamma_{\vec{v}}\|-\text{Env}$, $\rho \sqsubseteq \varrho$, and that in the evaluation of $T_0\rho_1$ the clock is bounded by K . Then $T_0\rho_1 \sqsubseteq \Phi_{d,K}(K - |a|)(\varrho[v_i \mapsto \text{val}(\varrho v_{ip})])$.

The Recomposition Lemma tells us that $\Phi_{d,K}(n)$ gives us a bound on the time-complexity of our recursion scheme. What we must do now is to “solve” the recurrence used to define Φ and show that it is polynomially-bounded.

THEOREM 3 (Bounding Lemma). *Suppose that in Theorem 1 we can assume that X and each Y_i are bounded by t.c. polynomials p and p_i , respectively. Assume further that p is $\langle\langle \mathbf{b} \rangle\rangle$ -safe and p_i is $\langle\langle \mathbf{b}_i \rangle\rangle$ -safe w.r.t. $\|\Gamma_{\vec{v}}\|$. Then there is a $\langle\langle \mathbf{b} \rangle\rangle$ -safe polynomial $\|\Gamma_{\vec{v}}\|, K : \langle\langle \mathbf{b}_1 \rangle\rangle, n : \langle\langle \mathbf{b}_1 \rangle\rangle \vdash \varphi(K, n) : \|\mathbf{b}\|$ such that for all K and n , $\Phi_{d,K}(n) \leq \varphi(K, n)$.*

Proof. Using the definition of d we can find a $\langle\langle \mathbf{b} \rangle\rangle$ -safe polynomial $\|\Gamma_{\vec{v}}\|, K : \langle\langle \mathbf{b}_1 \rangle\rangle \vdash (P_0(K), P_1) : \|\mathbf{b}\|$ and a recursive upper bound on $\Phi_{d,K}(n)\varrho$:

$$\begin{aligned} \Phi_{d,K}(0)\varrho &\leq (2K + 1, 0) \\ \Phi_{d,K}(n+1)\varrho &\leq (P_0(K), P_1)\varrho \uplus \text{pad}(\ell, \Phi_{d,K}(n)\varrho[v_i \mapsto \text{val}(p_{ip}\varrho)]) \end{aligned}$$

where $\ell = \text{tail_len}(f, t)$. An easy proof by induction shows that $\Phi_{d,K}(n) \leq (nP_0(K)\xi^{n-1} + 2K + 1, n\ell + P_1\xi^{n-1})$ for $n \geq 1$, where $\xi^0 = \text{id}$ and $(v_{ic}, v_{ip})\xi^{n+1} = \text{val}(p_{ip}\xi^n)$. Since $\ell \neq 0$ implies $\mathbf{b}_1 <: \mathbf{b}$, $n\ell + P_1\xi^{n-1}$ is bounded by a $\langle\langle \mathbf{b} \rangle\rangle$ -safe polynomial provided that $P_1\xi^{n-1}$ is $\langle\langle \mathbf{b} \rangle\rangle$ -safe. Since P_1 is $\langle\langle \mathbf{b} \rangle\rangle$ -safe and type-correct substitution of safe polynomials into a safe polynomial yields a safe polynomial (shown in Section 8 of *ATS*), to prove the theorem it suffices to show that $p_{ip}\xi^n$ is a $\langle\langle \mathbf{b}_i \rangle\rangle$ -safe polynomial for each i . The proof of this is essentially the proofs of the One-step and n -step lemmas of Section 10 in *ATS* (it is here that we use the remaining constraints on the types in the *crec* typing rule). \square

PROPOSITION 4 (Termination Lemma). *Assume the hypotheses of Theorem 3 hold and that $\rho \sqsubseteq \varrho$. Then in the evaluation of $T_0\rho_1$ the clock is bounded by $p_{1p}\xi^1\varrho$, where ξ^1 is defined as in the proof of Theorem 3.*

Proof. This follows from the details of the proof of Theorem 3. \square

THEOREM 5 (Soundness Theorem). *For every ATR term $\Gamma; \Delta \vdash t : \tau$ there is a $\text{tail}(\|\tau\|)$ -safe t.c. denotation X of type $\|\tau\|$ w.r.t. $\|\Gamma; \Delta\|$ such that $t \sqsubseteq X$.*

Proof. The proof is by induction on terms; for non-*crec* terms it is essentially as in *ATS*. For $\Gamma; _ \vdash \text{crec } a(\lambda_r f. \lambda \vec{v}. t) : \vec{\mathbf{b}} \rightarrow \mathbf{b}$, suppose $\tilde{\rho} \in \Gamma\text{-Env}$, $\tilde{\varrho} \in \|\Gamma\|\text{-Env}$, $\rho \sqsubseteq \varrho$. Use the Bounding, Termination and Recomposition Lemmas to show that $(\lambda \vec{v}. T_0)\tilde{\rho}_1 \sqsubseteq (\mathbb{A}_\star \vec{v}. \varphi(p_{1p}\xi^1, p_{1p}\xi^1 - |a|))\tilde{\varrho}$, where p_1 , φ , and ξ^n are as in the proof of the Bounding Lemma. We conclude that $\text{crec } a(\lambda_r f. \lambda \vec{v}. t) \sqsubseteq \text{dally}(1, \mathbb{A}_\star \vec{v}. \varphi(p_{1p}\xi^1, p_{1p}\xi^1 - |a|))$. Since this last time-complexity is a $\langle\langle \mathbf{b} \rangle\rangle$ -safe polynomial, the claim is proved. \square

COROLLARY 6. *If $_; _ \vdash t : \tau$, then t is computable in type-2 polynomial time.*

Soundness for recursion in an argument. We now address the recursions used in insertion-sort, in which the recursive use of the function occurs inside an argument to a previously-defined function. What we are really after here is structural (primitive) recursion for *defined* datatypes (such as our defined lists). First we adapt our \rightarrow -E rule to allow affine variables to appear in arguments to applications. We still require some restrictions in order to ensure a one-use property; the following is more than sufficient for our needs:

$$\frac{\Gamma; \Delta_0 \vdash s : \sigma \rightarrow \tau \quad \Gamma; \Delta_1 \vdash t : \sigma}{\Gamma; \Delta_0 \cup \Delta_1 \vdash st : \tau}$$

where at most one of Δ_0 and Δ_1 are non-empty, and if $level\ \sigma > 0$, then $\Delta_1 = \emptyset$. Thus the affine variable f may occur in either s or t but not both, and not t unless t is of base type; i.e., it is safe for β -reduction to copy a completed f -computation, but not an incomplete one. To simplify notation for the recursion present in insertion-sort we consider the special case in which we allow typings of the form $(*)$ provided $t = \text{if } s' \text{ then } s(f\vec{t}) \text{ else } s''$ where f is not free in s' or s'' (see Appendix D for a generalization).

First we must find a decomposition function. Assuming that $s \sqsubseteq X_s$, $t \sqsubseteq X_t$, and $t_i \sqsubseteq Y_i$, we can take as our decomposition function

$$d(\varrho, \chi) = X_t \varrho \uplus \left(cost(X_s \varrho) + cost(\chi \star \vec{X} \varrho) + cost(pot(X_s \varrho)(pot(\chi \star \vec{X} \varrho))), \right. \\ \left. pot(pot(X_s \varrho)(pot(\chi \star \vec{X} \varrho))) \right)$$

where we have written $\chi \star \vec{X} \varrho$ for $\chi \star X_1 \varrho \star \dots \star X_k \varrho$. Assume the inductively-given bounding t.c. denotations are bounded by safe polynomials p_s , p_t , and p_1, \dots, p_k . The Soundness Theorem follows from the Recomposition Lemma provided we have a polynomial bound on $\Phi_{d,K}(n)$, so now we establish such a bound.

When \mathbf{b} is oracular, then since $p_{sp} (= pot(p_s))$ is $\langle\langle \mathbf{b} \rangle\rangle$ -safe, we have that $p_{sp} = \lambda z^{\langle\langle \mathbf{b} \rangle\rangle}.(p, q_s \vee (r_s \vee z))$ where q_s is $\langle\langle \mathbf{b} \rangle\rangle$ -strict and r_s is $\langle\langle \mathbf{b} \rangle\rangle$ -chary and does not contain z . We can therefore find a $\langle\langle \mathbf{b} \rangle\rangle$ -safe t.c. polynomial $(P_0(K, z^{\langle\langle \mathbf{b} \rangle\rangle}), P_1)$ and derive the following recursive bound on $\Phi_{d,K}$ using the same conventions as in our analysis of cons-tail recursion:

$$\Phi_{d,K}(0) \varrho \leq (2K + 1, 0) \\ \Phi_{d,K}(n + 1) \varrho = (P_0(K, pot(\Phi_{d,K}(n) \varrho')), P_1) \uplus \Phi_{d,K}(n) \varrho'$$

where $\varrho' = \varrho[v_i \mapsto val(p_{ip} \varrho)]$. It is an easy induction to show that for $n \geq 1$ $\Phi_{d,K}(n) \leq ((n \cdot P_0(K, P_1) + 2K + 1)\xi^{n-1}, P_1 \xi^{n-1})$ and thus the Bounding and Termination Lemmas that must be proved are exactly those of before.

When \mathbf{b} is computational a similar calculation yields the bounding polynomial $((n \cdot P_0((n-2)q_s + P_1) + 2p_{1p})\xi^{n-1}, (n-1)q_s \xi^{n-2} + P_1 \xi^{n-1})$ for a $\langle\langle \mathbf{b} \rangle\rangle$ -strict polynomial q_s ; see Appendix D for details.

4. CONCLUDING REMARKS

In *ATS* we introduced the formalism *ATR* which captures the basic feasible functionals at type-level ≤ 2 . We have extended the formalism with recursion schemes that allow for more natural programming and demonstrated the new formalism by implementing lists of binary strings and insertion-sort and showing that the new recursion schemes do not take us out of the realm of feasibility. We have also given a strategy for proving that particular forms of recursion can be “safely” added to the base system. Here we indicate some future directions:

More general affine recursions. In Appendix D we give an inductive definition of *plain affine recursion* that generalizes cons-tail recursion, allows recursive calls in arguments, and permits recursive calls in the body of let-expressions. In particular, it covers all forms of recursion used in the list operations and insertion- and selection-sort. At the time of writing, we do not have all the details of the soundness argument in the general case, but we expect it to follow the framework we have developed here. The main addition looks to be an inductive definition of the decomposition functions (the decomposition function we used in our analysis here is the unwinding of this definition to the special case at hand). The key point of the framework is that it gives us a soundness proof relative to the decomposition functions.

Lazy ATR. A version of ATR with lazy constructors (streams) and evaluation would be very interesting. There are many technical challenges in analyzing such a system but again we expect that the general outline will be the approach we have used in this paper. Of course one can implement streams in the current call-by-value setting in standard ways (raising the type-level), but a direct lazy implementation of streams is likely to be more informative. We expect the analysis of such a lazy-ATR to require an extensive reworking of the various semantic models we have discussed here and in *ATS*.

Real-number algorithms. ATR is a type-2 language, but here we have focused on type-1 algorithms. We are working on implementing real-number algorithms, viewing a real number as a type-1 (stream) oracle. This can be done in either a call-by-value setting (e.g., algorithms that take a string of length n as input and return something like an n -bit approximation of the result) or a lazy setting (in which the algorithm returns bits of the result on demand).

REFERENCES

- [1] K. Aehlig and H. Schwichtenberg. A syntactical analysis of non-size-increasing polynomial time computation. *ACM Transactions on Computation Logic*, 3(3):383–401, 2002. URL <http://doi.acm.org/10.1145/507382.507386>.
- [2] A. Barber. Dual intuitionistic linear logic. Technical Report ECS-LFCS-96-347, Laboratory for Foundations of Computer Science, 1996. URL <http://www.lfcs.inf.ed.ac.uk/reports/96/ECS-LFCS-96-347/index.html>.
- [3] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational Complexity*, 2(2):97–110, 1992. URL <http://dx.doi.org/10.1007/BF01201998>.
- [4] S. Cook and A. Urquhart. Functional interpretations of feasibly constructive arithmetic. *Annals of Pure and Applied Logic*, 63(2):103–200, 1993. URL [http://dx.doi.org/10.1016/0168-0072\(93\)90044-E](http://dx.doi.org/10.1016/0168-0072(93)90044-E).
- [5] N. Danner and J. S. Royer. Adventures in time and space. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Charleston, SC, 2006)*, pages 168–179, New York, 2006. Association for Computing Machinery. URL <http://doi.acm.org/10.1145/1111037.1111053>.
- [6] N. Danner and J. S. Royer. Adventures in time and space. URL <http://arxiv.org/abs/cs/0612116>. To appear in *Logical Methods in Computer Science*.
- [7] C. C. Frederiksen and N. D. Jones. Recognition of polynomial-time programs. Technical Report TOPPS/D-501, DIKU, University of Copenhagen, 2004. URL <http://www.diku.dk/topps/bibliography/2004.html>.
- [8] M. Hofmann. Linear types and non-size-increasing polynomial time computation. *Information and Computation*, 183(1):57–85, 2003. URL [http://dx.doi.org/10.1016/S0890-5401\(03\)00009-9](http://dx.doi.org/10.1016/S0890-5401(03)00009-9).
- [9] M. Hofmann. The strength of non-size increasing computation. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Portland, OR, 2002)*, pages 260–269, New York, 2002. ACM Press. URL <http://doi.acm.org/10.1145/503272.503297>.
- [10] N. D. Jones. The expressive power of higher-order types or, life without cons. *Journal of Functional Programming*, 11(1):55–94, 2001. URL <http://dx.doi.org/10.1017/S0956796800003889>.

$$\begin{aligned}
s, t &::= V \mid K \mid O \\
&\mid (c_a s) \mid (d s) \mid (t_a s) \mid (\text{if } s \text{ then } t_0 \text{ else } t_1) \mid (\text{down } s t) \mid (\text{crec } K(\lambda_r f.t)) \\
&\mid (\lambda V.s) \mid (st) \\
K &::= \{0, 1\}^*
\end{aligned}$$

FIGURE 2. ATR expressions. V is a set of variable symbols and O a set of oracle symbols.

- [11] D. Leivant. Ramified recurrence and computational complexity I: Word recurrence and poly-time. In *Feasible Mathematics II (Ithaca, NY, 1992)*, pages 320–343. Birkhäuser Boston, Boston, MA, 1995.
- [12] K. Mehlhorn. Polynomial and abstract subrecursive classes. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing (Seattle, WA, 1974)*, pages 96–109, New York, NY, USA, 1974. ACM Press. URL <http://doi.acm.org/10.1145/800119.803890>.

APPENDIX A. TYPING RULES AND EVALUATION

Recall that labels L are elements of $(\Box\Diamond)^* \cup \Diamond(\Box\Diamond)^*$. We define $\Box_0 = \varepsilon$, $\Diamond_d = \Diamond\Box_d$, and $\Box_{d+1} = \Box\Diamond_d$. We give the ATR expressions and typing rules in Figures 2 and 3.⁸ For convenience, we view oracle symbols as different syntactic objects than (type-level-1) variables; essentially they are variables with a fixed meaning and that cannot be abstracted. The idea behind the (**Shift**) rule is as follows. Suppose one has a function f of type $\mathbf{N}_\varepsilon \rightarrow \mathbf{N}_\Diamond$; the types tell us that the function performs some polynomial-time computation on its input. Intuitively, we should be able to apply f to an input of any type \Box_d ; f then applies some polynomial-time computation to its input, yielding an output of type $\Diamond\Box_d = \Diamond_d$. A special case of the *shifts-to* relation is that $\Box_0 \rightarrow \Diamond_0 \propto \Box_d \rightarrow \Diamond_d$ for any d . The full definition (see *ATS*) of \propto extends this idea to all types of level ≤ 2 .

We define the evaluation relation in Figure 4. This relates closures to values, defined simultaneously as follows:

- (1) A *closure* $t\rho$ consists of a term and an environment such that every free variable of t is in the domain of ρ and for every x in the domain of ρ , $\rho(x)$ is a closure.
- (2) A *value* $z\theta$ is a closure in which z is either a string constant, oracle, or abstraction.
- (3) An *extended value* $z\theta$ is a closure that is either a value or for which $z = \text{crec } a(\lambda_r f.\lambda \vec{v}.t)$ for some string constant a , variables f and \vec{v} , and term t .
- (4) An environment is a finite map from term variables to extended values.

Recalling that oracles range over type-1 functions and that the only type-0 values are string constants, the evaluation rules O_0 and O_1 says to treat multiple-argument oracles as though they are in curried form, returning the curried oracle result until all arguments have been provided. The cost of each rule is 1 with the following exceptions:

- (1) The cost of (Env) is $1 \vee |z|$ if z is a string constant and 1 otherwise;
- (2) The cost of (down_i) is $2|K_t| + 1$;
- (3) The cost of (O_0) is $|K| + 1$ and the cost of (O_1) is 1.

These costs reflect a length-cost model of accessing the environment or evaluating an oracle and an evaluation of $|K_s| \leq |K_t|$ by stripping off bits one-by-one from each of K_s and K_t .

The typing rules for t.c. polynomials are given in Figure 5.

⁸In *ATS*, we restricted to tail-recursion and thus needed no constraint on \mathbf{b}_0 in the (**crec-I**) rule; we have not seen any natural programs in which this constraint is violated.

$$\begin{array}{c}
\textbf{Zero-I} \frac{}{\Gamma; \Delta \vdash \varepsilon : N_\varepsilon} \quad \textbf{Const-I} \frac{}{\Gamma; \Delta \vdash K : N_\diamond} \\
\textbf{Int-Id-I} \frac{}{\Gamma, v : \sigma; \Delta \vdash v : \sigma} \quad \textbf{Aff-Id-I} \frac{}{\Gamma; \Delta, v : \sigma \vdash v : \sigma} \\
\textbf{Shift} \frac{\Gamma; \Delta \vdash s : \sigma}{\Gamma; \Delta \vdash s : \tau} (\sigma \propto \tau) \quad \textbf{Subsumption} \frac{\Gamma; \Delta \vdash s : \sigma}{\Gamma; \Delta \vdash s : \tau} (\sigma \leq \tau) \\
\textbf{c}_a\textbf{-I} \frac{\Gamma; \Delta \vdash s : N_{\diamond_d}}{\Gamma; \Delta \vdash (\textbf{c}_a s) : N_{\diamond_d}} \quad \textbf{d-I} \frac{\Gamma; \Delta \vdash s : N_L}{\Gamma; \Delta \vdash \textbf{d} s : N_L} \quad \textbf{t}_a\textbf{-I} \frac{\Gamma; \Delta \vdash s : N_L}{\Gamma; \Delta \vdash \textbf{t}_a s : N_L} \\
\textbf{down-I} \frac{\Gamma; \Delta_0 \vdash s : N_{L_0} \quad \Gamma; \Delta_1 \vdash t : N_{L_1}}{\Gamma; \Delta_0, \Delta_1 \vdash (\textbf{down} st) : N_{L_1}} \\
\textbf{if-I} \frac{\Gamma; _ \vdash s : N_L \quad \Gamma; \Delta_0 \vdash t_0 : N_{L'} \quad \Gamma; \Delta_1 \vdash t_1 : N_{L'}}{\Gamma; \Delta_0 \cup \Delta_1 \vdash (\textbf{if} s \textbf{ then } t_0 \textbf{ else } t_1) : N_{L'}} \\
\textbf{crec-I} \frac{_ \vdash K : N_\diamond \quad \Gamma, \vec{v} : \vec{b}; f : \vec{b} \rightarrow b_0 \vdash t : b_0}{\Gamma; _ \vdash \textbf{crec} a (\lambda_r f. \lambda \vec{v}. t) : \vec{b} \rightarrow b_0} \\
\textbf{\(\rightarrow\)-I} \frac{\Gamma, v : \sigma; \Delta \vdash t : \tau}{\Gamma; \Delta \vdash (\lambda v. t) : \sigma \rightarrow \tau} \quad \textbf{\(\rightarrow\)-E} \frac{\Gamma; \Delta \vdash s : \sigma \rightarrow \tau \quad \Gamma; _ \vdash t : \sigma}{\Gamma; \Delta \vdash (st) : \tau}
\end{array}$$

FIGURE 3. ATR typing. In **crec-I**, $\vec{b} = b_1, \dots, b_k$, $b_i \leq b_1$ implies b_i is oracular (including $i = 0$), and f is in cons-tail position in t . The changes from *ATS* are as follows: (1) *ATS* imposed no constraint on b_0 in (**crec-I**); (2) *ATS* restricted (**crec-I**) to tail-recursion; and (3) *ATS* restricted (**d-I**) and (**t_a-I**) to computational types.

APPENDIX B. CODE FOR LISTS AND SORTING

Code for basic list operations is given in Figure 6 and for selection-sort in Figure 7. The function *leq* used in Figure 7 tests two integers written in binary for inequality; we leave its full definition as an exercise for the reader. Recall that if x then y else z evaluates to y if $x \neq \varepsilon$ and z otherwise.

APPENDIX C. PROOFS OF THE MAIN THEOREMS

In this section, we prove the Recomposition Lemma (Theorem 2). As a guide to the notation, environments ρ and ϱ typically refer to $\Gamma_{\vec{v}}$ and $\|\Gamma_{\vec{v}}\|$ environments and environments $\tilde{\rho}$ and $\tilde{\varrho}$ typically refer to Γ and $\|\Gamma\|$ -environments.

First we formalize the notion of “hard-coding” an upper bound for the clock. Note that to evaluate $\textbf{crec} a (\lambda_r f. \lambda \vec{v}. t)$ applied to appropriate arguments, we really evaluate $T_0 \rho_1$. Suppose we have a typing of the form $(*)$ and consider the evaluation of $T_\ell \rho_{\ell+1}$ where we assume that the **crec** clock-test does not terminate the recursion. The evaluation has the form:

$$\frac{\frac{\frac{(0^\ell a) \rho_{\ell+1} \downarrow (0^\ell a) []}{(c_0(0^\ell a)) \rho_{\ell+1} \downarrow (0^{\ell+1} a) []} \quad \frac{v_1 \rho_{\ell+1} \downarrow \dots}{(c_0(v_1)) \rho_{\ell+1} \downarrow \dots}}{\frac{(\textbf{down}(c_0(0^\ell a))(c_0 v_1)) \rho_{\ell+1} \downarrow (0^{\ell+1} a) []}{(T_\ell) \rho_{\ell+1} \downarrow \dots}} \quad \mathcal{D} \quad t \rho_{\ell+1} \downarrow \dots$$

where \mathcal{D} is the derivation

$$\begin{array}{c}
\overline{z\theta \downarrow z\theta} \text{ (} z\theta \text{ a value)} \\
\hline
(\text{crec } a (\lambda_r f. \lambda \vec{v}. t)) \rho \downarrow (\lambda \vec{v}. \text{if } |a| < |v_1| \text{ then } t \text{ else } \varepsilon) \rho [f \mapsto \text{crec}(\mathbf{0}a)(\lambda_r f \lambda \vec{v}. t)] \\
\hline
\mathbf{Env} \frac{\rho(x) \downarrow z\theta}{x\rho \downarrow z\theta} \quad \frac{s\rho \downarrow K\theta}{(\mathbf{c}_a s)\rho \downarrow (aK)\theta} \\
\hline
\frac{s\rho \downarrow \varepsilon\theta}{(\mathbf{d} s)\rho \downarrow \varepsilon\theta} \quad \frac{s\rho \downarrow (aK)\theta}{(\mathbf{d} s)\rho \downarrow K\theta} \quad \frac{s\rho \downarrow (aK)\theta}{(\mathbf{t}_a s)\rho \downarrow \mathbf{0}\theta} \quad \frac{s\rho \downarrow K\theta}{(\mathbf{t}_a s)\rho \downarrow \varepsilon[]} \text{ (} K \neq aK' \text{ any } K') \\
\hline
\text{down}_0 \frac{s\rho \downarrow K_s \theta_s \quad t\rho \downarrow K_t \theta_t \quad |K_s| \leq |K_t|}{(\text{down } st)\rho \downarrow K_s \theta_s} \\
\hline
\text{down}_1 \frac{s\rho \downarrow K_s \theta_s \quad t\rho \downarrow K_t \theta_t \quad |K_s| > |K_t|}{(\text{down } st)\rho \downarrow \varepsilon[]} \\
\hline
\frac{s\rho \downarrow (aK)\theta \quad t_0\rho \downarrow z\theta}{(\text{if } s \text{ then } t_0 \text{ else } t_1)\rho \downarrow z\theta} \quad \frac{s\rho \downarrow \varepsilon\theta \quad t_1\rho \downarrow z\theta}{(\text{if } s \text{ then } t_0 \text{ else } t_1)\rho \downarrow z\theta} \\
\hline
\frac{s\rho \downarrow (\lambda x. s')\theta' \quad t\rho \downarrow z\theta \quad s'\theta'[x \mapsto z\theta] \downarrow v\eta}{(st)\rho \downarrow v\eta} \\
\hline
O_0 \frac{s\rho \downarrow O\theta' \quad t\rho \downarrow z\theta \quad O(\llbracket z \rrbracket) = K}{(st)\rho \downarrow K[]} \\
\hline
O_1 \frac{s\rho \downarrow O\theta' \quad t\rho \downarrow z\theta \quad O(\llbracket z \rrbracket) = O'}{(st)\rho \downarrow O'[]}
\end{array}$$

FIGURE 4. ATR evaluation.

$$\begin{array}{c}
\overline{\Sigma \vdash \varepsilon : \mathbb{T}_\varepsilon} \quad \overline{\Sigma \vdash \mathbf{0}^n : \mathbb{T}_\diamond} \quad \overline{\Sigma, x : \gamma \vdash x : \gamma} \\
\hline
\frac{\Sigma \vdash p : \gamma}{\Sigma \vdash p : \gamma'} (\gamma \propto \gamma') \quad \frac{\Sigma \vdash p : \gamma}{\Sigma \vdash p : \gamma'} (\gamma \leq \gamma') \\
\hline
\frac{\Sigma \vdash p : \mathbb{T}_{\diamond_k} \quad \Sigma \vdash q : \mathbb{T}_{\diamond_k}}{\Sigma \vdash p \bullet q : \mathbb{T}_{\diamond_k}} \quad \frac{\Sigma \vdash p : \gamma \quad \Sigma \vdash q : \gamma}{\Sigma \vdash p \vee q : \gamma} \\
\hline
\frac{\Sigma, x : \sigma \vdash p : \tau}{\Sigma \vdash \lambda x. p : \sigma \rightarrow \tau} \quad \frac{\Sigma \vdash p : \sigma \rightarrow \tau \quad \Sigma \vdash q : \sigma}{\Sigma \vdash pq : \tau}
\end{array}$$

FIGURE 5. Typing rules for t.c. polynomials. \bullet is $+$ or $*$, γ is a t.c. base type, and $\gamma \leq \gamma'$ is defined by $\mathbb{T}_{\square_k} \leq \mathbb{T}_{\diamond_k} \leq \mathbb{T}_{\square_{k+1}}$ and $\mathbb{T}_L \leq \mathbb{T}$ for all L .

$$\begin{array}{c}
\overline{(C_{\ell+1})\rho \downarrow (\lambda \vec{v}. T_{\ell+1})\rho_{\ell+2}} \\
\hline
\overline{f\rho_{\ell+1} \downarrow (\lambda \vec{v}. T_{\ell+1})\rho_{\ell+2}} \\
\hline
\vdots \quad \frac{\vdots \quad \overline{(T_{\ell+1})\rho_{\ell+2} [v_i \mapsto z_i \theta_i] \downarrow \dots}}{t_k \rho_{\ell+1} \downarrow z_k \theta_k} \quad \vdots \\
\hline
\overline{(f\vec{t})\rho_{\ell+1} \downarrow \dots} \\
\hline
\vdots \\
\hline
\overline{t\rho_{\ell+1} \downarrow \dots}
\end{array}$$

```

val nil = ε : Nε

val cons : Nε → N◇ → N◇ =
  fn w l ⇒ letrec enc : Nε → N◇ → N◇ =
    fn b x ⇒ if x then if t0(x) then c1(c0(enc b (d x)))
                  else c1(c1(enc b (d x)))
                else c0(l)
  in enc w w end

val head : Nε → Nε =
  fn l ⇒ letrec dec : Nε → N◇ → N◇ =
    fn b x ⇒ if t1(x) then
      if t0(d x) then c0(dec b (d(d(x)))) else c1(dec b (d(d(x))))
    else ε
  in down (dec l l)(l) end

val tail : Nε → Nε =
  fn l ⇒ letrec tail' : Nε → Nε → Nε =
    fn b x ⇒ if t1(x) then tail' b d(d(x)) else d(x)
  in tail' l l end

```

FIGURE 6. Primitive list operations in ATR.

```

val g : Nε → Nε → N◇ =
  fn y z ⇒ if leq y (head z) then cons y z
            else cons (head z) (cons y (tail z))

val select : Nε → Nε =
  fn l ⇒ letrec sel : Nε → Nε → Nε =
    fn b l' ⇒ if tail(l') then down (g (head l') (sel b (tail l'))) l' else l'
  in sel l l end

val selSort : Nε → N◇ =
  fn l ⇒ letrec ssort : Nε → Nε → N◇ =
    fn b l' ⇒ let
      val m = select l'
    in
      cons (head m) (ssort b (tail m))
    end
  in ssort l l end

```

FIGURE 7. Selection-sort in ATR. Note: let val $x=s$ in t end abbreviates $(\text{fn } x \Rightarrow t)s$ where we restrict x to be of base type.

provided that $t\rho_{\ell+1}$ actually makes a recursive call. Thus we see that all closures over some T_m in the evaluation of $T_\ell\rho_{\ell+1}$ have the form $T_m\rho_{m+1}[\vec{v} \mapsto \vec{z}\vec{\theta}]$. For a particular closure $T_\ell\rho_{\ell+1}$ we say that the *clock is bounded by K* if in its evaluation, for every subevaluation of a closure $T_m\rho_{m+1}[\vec{v} \mapsto \vec{z}\vec{\theta}]$ it is the case that $|z_1| < K$.

To prove the Recomposition Lemma, we embed the evaluation of a clocked recursion in which the clock is bounded into an evaluation in which the clock is fixed. To this end, introduce new term constructors rec_K with the following evaluation rule:

$$(\text{rec}_K a(\lambda_r f.\lambda \vec{v}.t))\rho \downarrow (\lambda \vec{v}.\text{if } |a| < |0^K| \text{ then } t \text{ else } \varepsilon)\rho[f \mapsto (\text{rec}_K(0a)(\lambda_r f.\lambda \vec{v}.t))]$$

Set

$$C_{K,\ell} = \text{rec}_K(0^\ell a)(\lambda_r f.\lambda \vec{v}.t) \quad T_{K,\ell} = \text{if } |0^\ell a| < |0^K| \text{ then } t \text{ else } \varepsilon$$

and for an environment ρ set $\rho_{K,\ell} = \rho[f \mapsto C_{K,\ell}]$.

LEMMA 7. *Suppose that whenever $\tilde{\rho} \in (\Gamma; f : \gamma)\text{-Env}$, $\tilde{\varrho} \in \|\Gamma\|\text{-Env}$, and $\tilde{\rho} \upharpoonright \text{Dom } \Gamma \sqsubseteq \tilde{\varrho}$, it is the case that $(\lambda v.t)\tilde{\rho} \sqsubseteq (\mathbb{A}_* v.X)\tilde{\varrho}$. If $\rho \in (\Gamma_{\vec{v}}; f : \gamma)\text{-Env}$, $\varrho \in \|\Gamma_{\vec{v}}; f : \gamma\|\text{-Env}$, and $\rho \upharpoonright \text{Dom } \Gamma_{\vec{v}} \sqsubseteq \varrho$, then $t\rho \sqsubseteq X(\varrho[v_i \mapsto \text{val}(\varrho v_{ip})])$.*

DEFINITION 5. For $\Phi_{d,K}$ as defined in Section 3, define $\tilde{\Phi}_{d,K}(n) = \mathbb{A}_* \vec{v}.\Phi_{d,K}(n)$.

DEFINITION 6. For a t.c. environment ϱ defined on $\|\vec{v}\|$, define $\varrho^V = \varrho[v_i \mapsto \text{val}(\varrho v_{ip})]$.

LEMMA 8. *Suppose $\Gamma, \vec{v} : \vec{\mathbf{b}}; f : \vec{\mathbf{b}} \rightarrow \mathbf{b} \vdash t : \mathbf{b}$ and that d is a decomposition function for t .*

- (1) *Suppose $\tilde{\rho} \in \Gamma\text{-Env}$, $\tilde{\varrho} \in \|\Gamma\|\text{-Env}$, and $\tilde{\rho} \sqsubseteq \tilde{\varrho}$. Then $(\lambda \vec{v}.T_{K,\ell})\tilde{\rho}_{K,\ell+1} \sqsubseteq \tilde{\Phi}_{d,K}(K - |0^\ell a|)\tilde{\varrho}$.*
- (2) *Suppose $\rho \in \Gamma_{\vec{v}}\text{-Env}$, $\varrho \in \|\Gamma_{\vec{v}}\|\text{-Env}$, $\rho \sqsubseteq \varrho$. Then $T_{K,\ell}\rho_{K,\ell+1} \sqsubseteq \Phi_{d,K}(K - |0^\ell a|)\varrho^V$.*

Proof. The second part follows from the first by Lemma 7, so we just prove the first by induction on $K - |0^\ell a|$. The base case is immediate. The induction hypothesis tells us that $(\lambda \vec{v}.T_{K,\ell+1})\tilde{\rho}_{K,\ell+2} \sqsubseteq \tilde{\Phi}_{d,K}(K - |0^\ell a| - 1)\tilde{\varrho}$. Set $\tilde{\varrho}(f) = \text{dally}(2, \tilde{\Phi}_{d,K}(K - |0^\ell a| - 1)\tilde{\varrho})$. Then since $f\rho_{K,\ell+1}$ evaluates to $(\lambda \vec{v}.T_{K,\ell+1})\rho_{K,\ell+2}$ in two steps, we have that $f\tilde{\rho}_{K,\ell+1} \sqsubseteq \text{dally}(2, \tilde{\Phi}_{d,K}(K - |0^\ell a| - 1)\tilde{\varrho}) = \tilde{\varrho}(f)$ and thus $\tilde{\rho}_{K,\ell+1} \sqsubseteq \tilde{\varrho}$. Since d is a decomposition function for t , we have that

$$\begin{aligned} (\lambda \vec{v}.T_{K,\ell})\tilde{\varrho}_{K,\ell+1} &= (\mathbb{A}_* \vec{v}.\mathbb{A}\varrho. \text{dally}(2K+1, d(\varrho_\varepsilon, \varrho f) \vee (1, 0)))\tilde{\varrho} \\ &= (1, \mathbb{A}v_{1p}(\dots (1, \mathbb{A}v_{kp}. \text{dally}(2K+1, \\ &\quad d(\tilde{\varrho}_\varepsilon[v_i \mapsto \text{val}(v_{ip})], \text{dally}(2, \tilde{\Phi}_{d,K}(K - |0^\ell a| - 1)\tilde{\varrho})) \vee \\ &\quad (1, 0))) \dots)) \\ &= (1, \mathbb{A}v_{1p}(\dots (1, \mathbb{A}v_{kp}. \text{dally}(2K+1, \\ &\quad d(\tilde{\varrho}_\varepsilon[v_i \mapsto \text{val}(v_{ip})], \\ &\quad \text{dally}(2, \tilde{\Phi}_{d,K}(K - |0^\ell a| - 1)\tilde{\varrho}[v_i \mapsto \text{val}(v_{ip})])) \vee \\ &\quad (1, 0))) \dots)) \\ &= (\mathbb{A}_* \vec{v}.\mathbb{A}\varrho. \text{dally}(2K+1, \\ &\quad d(\varrho_\varepsilon, \text{dally}(2, \tilde{\Phi}_{d,K}(K - |0^\ell a| - 1)\varrho)) \vee (1, 0)))\tilde{\varrho} \\ &= (\mathbb{A}_* \vec{v}.\Phi_{d,K}(K - |0^\ell a|))\tilde{\varrho} \\ &= \tilde{\Phi}_{d,K}(K - |0^\ell a|)\tilde{\varrho}. \end{aligned}$$

□

THEOREM 9 (Theorem 2: Recomposition Lemma). *Assume the hypotheses of Lemma 8(2). Assume further that in the evaluation of $T_{0\rho_1}$ the clock is bounded by K . Then $T_{0\rho_1} \sqsubseteq \Phi_{d,K}(K - |a|)\varrho^V$.*

Proof Sketch. The hypotheses allow us to define an injective map F from the evaluation derivation of $T_{0\rho_1}$ to the evaluation derivation of $T_{K,0\rho_{K,1}}$ such that:

- (1) F maps the root to the root;
- (2) F preserves the “child-of” relation;
- (3) The only differences between the closures at the node x and $F(x)$ are:
 - (a) C_m is replaced with $C_{K,m}$;
 - (b) T_m is replaced with $T_{K,m}$;
 - (c) The evaluations of $(\text{down}(\text{c}_0(0^m a)(\text{c}_0 v_1)))\rho'_{m+1}$ are mapped to evaluations of $(\text{down}(\text{c}_0(0^m a)(\text{c}_0 0^K)))\rho'_{m+1}$.

Thus we have that the evaluation derivation of $T_0\rho_1$ is no larger than that of $T_{K,0}\rho_{K,1}$ and that $T_0\rho_1 \downarrow z\theta$ iff $T_{K,0}\rho_{K,1} \downarrow z\theta$. From this we conclude that since $(T_{K,0})\rho_{K,1} \sqsubseteq \Phi_{d,K}(K - |a|)\varrho^V$ we also have that $(T_0)\rho_1 \sqsubseteq \Phi_{d,K}(K - |a|)\varrho^V$. \square

THEOREM 10 (Theorem 5: Soundness Theorem). *If $\Gamma; \Delta \vdash t : \tau$ is an ATR term, then there is a tail($\|\tau\|$)-safe t.c. denotation X of type $\|\tau\|$ w.r.t. $\|\Gamma; \Delta\|$ such that $t \sqsubseteq X$.*

Proof. The proof is by induction on t . For everything but **crec** terms, it is mostly a pushing-through of the definition of \sqsubseteq . Now suppose that $\Gamma; _ \vdash \text{crec } a(\lambda_r f. \lambda \vec{v}. t) : \vec{b} \rightarrow \mathbf{b}$, $\tilde{\rho} \in \Gamma\text{-Env}$, $\tilde{\varrho} \in \|\Gamma\|\text{-Env}$, and that $\tilde{\rho} \sqsubseteq \tilde{\varrho}$. Noting that $(\text{crec } a(\lambda_r f. \lambda \vec{v}. t))\tilde{\rho} \downarrow (\lambda \vec{v}. T_0)\tilde{\rho}_1$, we wish to show that this latter term is bounded by $(\mathbb{A}_\star \vec{v}. \varphi(p_{1p}\xi^1, p_{1p}\xi^1 - |a|, \vec{v}))\tilde{\varrho}$ where φ and ξ are as in the proof of the Bounding Lemma. To do so, it suffices to show that if $z_i\theta_i \sqsubseteq_{\text{pot}} q_i$, $\tilde{\rho}_1^* = \tilde{\rho}_1[v_i \mapsto z_i\theta_i]$, and $\tilde{\varrho}^* = \tilde{\varrho}[v_i \mapsto \text{val}(q_i)]$, then $T_0\tilde{\rho}_1^* \sqsubseteq \varphi(p_{1p}\xi^1, p_{1p}\xi^1 - |a|, \vec{v})\tilde{\varrho}^*$. Since $\tilde{\rho}_1^* \sqsubseteq \tilde{\varrho}^*$, from the Termination Lemma we have that the clock on $T_0\tilde{\rho}_1^*$ is bounded by $p_{1p}\xi^1\tilde{\varrho}^*$. Thus by the Recomposition Lemma we have that

$$T_0\tilde{\rho}_1^* \sqsubseteq \Phi_{d,p_{1p}\xi^1\tilde{\varrho}^*}(p_{1p}\xi^1\tilde{\varrho}^* - |a|)(\tilde{\varrho}^*)^V = \Phi_{d,p_{1p}\xi^1\tilde{\varrho}^*}(p_{1p}\xi^1\tilde{\varrho}^* - |a|)\tilde{\varrho}^* \leq \varphi(p_{1p}\xi^1, p_{1p}\xi^1 - |a|, \vec{v})\tilde{\varrho}^*.$$

We conclude that $(\text{crec } a(\lambda_r f. \lambda \vec{v}. t))\tilde{\rho} \sqsubseteq \text{dally}(1, \mathbb{A}_\star \vec{v}. \varphi(p_{1p}\xi^1, p_{1p}\xi^1 - |a|, \vec{v}))\tilde{\varrho}$ and hence that $\text{crec } a(\lambda_r f. \lambda \vec{v}. t) \sqsubseteq \text{dally}(1, \mathbb{A}_\star \vec{v}. \varphi(p_{1p}\xi^1, p_{1p}\xi^1 - |a|, \vec{v}))$. \square

APPENDIX D. PLAIN AFFINE RECURSION

We generalize the recursion schemes we have discussed in this paper as follows:

DEFINITION 7. t is a *plain affine recursive definition* of f if:⁹

- (1) $f \notin \text{fv}(t)$; or
- (2) $t = ft_1 \dots t_k$ where $f \notin \text{fv}(t_i)$ for any i ;
- (3) $t = \text{if } s \text{ then } s_0 \text{ else } s_1$ where $f \notin \text{fv}(s)$ and each s_i is a plain affine recursive definition of f ;
- or
- (4) $t = \text{op } s$ where **op** is any of c_a , **d**, or t_a and s is a plain affine recursive definition of f ; or
- (5) $t = \text{down } s_0 s_1$ where s_0 is a plain affine recursive definition of f and $f \notin \text{fv}(s_1)$; or
- (6) $t = st_1 \dots t_k$ where $f \notin \text{fv}(s)$ and each t_i is a plain affine recursive definition of f ; or
- (7) $t = (\lambda x.s)r$ where s is a plain affine recursive definition of f and $f \notin \text{fv}(r)$.

We continue here to consider the special case of $t = \text{if } s' \text{ then } s(\vec{f}\vec{t}) \text{ else } s''$ where f is not free in s' or s'' . We have already established a decomposition function; all that remains to to set up and solve a recursive bound on $\Phi_{d,K}(n)$ when \mathbf{b} is computational. In this case $p_{sp} = \lambda z^{\langle\langle \mathbf{b} \rangle\rangle}.(p, q_s + (r_s \vee z))$ where q_s is $\langle\langle \mathbf{b} \rangle\rangle$ -strict and r_s is $\langle\langle \mathbf{b} \rangle\rangle$ -chary and does not contain z . The recurrence to solve is

$$\begin{aligned} \Phi_{d,K}(0)\varrho &\leq (2K + 1, 0) \\ \Phi_{d,K}(n + 1)\varrho &\leq (P_0(K, \text{pot}(\Phi_{d,K}(n)\varrho')), P_1) \uplus \text{pad}(q_s, \Phi_{d,K}(n)\varrho') \end{aligned}$$

⁹Clearly we are duplicating work that the affine type system does for us here; we have not yet fully investigated this situation.

where $\varrho' = \varrho[v_i \mapsto \text{val}(p_{ip}\varrho)]$ and $(P_0(K, z^{\langle\mathbf{b}\rangle}), P_1)$ is a $\langle\mathbf{b}\rangle$ -safe t.c. polynomial. The solution to this recurrence is given by

$$\Phi_{d,K}(n) \leq ((n \cdot P_0((n-2)q_s + P_1) + 2K + 1)\xi^{n-1}, (n-1)q_s\xi^{n-2} + P_1\xi^{n-1})$$

for $n \geq 2$, so the Bounding and Terminations Lemmas to be proved are those of before. Furthermore, since \mathbf{b} is computational, $\mathbf{b} > \mathbf{b}_1$ and so we have that $n^{\langle\mathbf{b}_1\rangle}q_s$ is a \mathbf{b} -strict polynomial, and hence $nq_s\xi^{n-1} + P_1\xi^n$ is \mathbf{b} -safe for each n . The rest of the Soundness Theorem follows.

DEPARTMENT OF MATHEMATICS AND COMPUTER SCIENCE, WESLEYAN UNIVERSITY, MIDDLETOWN, CT 06459, USA

E-mail address: `ndanner@wesleyan.edu`

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE, SYRACUSE UNIVERSITY, SYRACUSE, NY 13210, USA

E-mail address: `royer@ecs.syr.edu`