Termination Analysis based on Operational Semantics

Flemming Nielson, Hanne Riis Nielson
Computer Science Department, Aarhus University
Ny Munkegade, DK-8000 Aarhus C, Denmark
e-mail:{fnielson,hrnielson}@daimi.aau.dk

Abstract

In principle termination analysis is easy: find a well-founded partial order and prove that calls decrease with respect to this order. In practice this often requires an *oracle* (or a theorem prover) for determining the well-founded order and this oracle may not be easily implementable. Our approach circumvents some of these problems by exploiting the inductive definition of algebraic data types and using pattern matching as in functional languages. We develop a termination analysis for a higher-order functional language; the analysis incorporates and extends polymorphic type inference and axiomatizes a class of well-founded partial orders for *multiple-argument* functions (as in Standard ML and Miranda).

Semantics is given by means of operational (natural-style) semantics and soundness is proved; this involves making extensions to the semantic universe and we relate this to the techniques of denotational semantics. For dealing with the partiality aspects of the soundness proof it suffices to incorporate approximations to the desired fixed points; for dealing with the totality aspects of the soundness proof we also have to incorporate functions that are forced to terminate (in a way that might violate the *monotonicity* of denotational semantics).

Keywords: Semantics of programming languages, program analysis by annotated type systems, proof techniques for operational semantics.

1 Introduction

Total correctness of programs is often treated as two separate problems: one is *partial correctness* saying that if the program terminates then the result satisfies its specification and the other is *termination* that ensures that the program does terminate. Techniques for analysing programs so as to ensure termination are therefore valuable for program development and for being able to guarantee the correctness of using (possibly partial) program-defined functions in expressing semantic conditions such as invariants.

For another example closer to programming consider a multi-paradigmatic programming language that allows communication. A good design principle is that processes should spend as little time in critical regions as possible and surely it is unacceptable if a process might loop or fail while inside the critical region. Similarly, the enforcement of fairness among processes being multi-tasked on a single processor might be facilitated by not having to call the scheduler within subcomputations known to terminate within a reasonable time bound. (In the Conclusion we briefly discuss the possibility of extending our approach with time-complexity.).

Our approach is motivated by the belief that it may be beneficial to *extend type systems* with special notations for the type of functions that can be guaranteed to terminate. While there are other approaches to the same goal we believe that a main advantage of the approach based on type systems is to make the findings of the analysis available to the programmer; indeed it could form an important part of the interface definitions of modules.

The present paper demonstrates that such an approach is semantically sound and briefly discusses how to modify the standard type inference algorithm so as to obtain an implementation. Since the underlying problem is undecidable this means that our approach must have some weaknesses; yet we believe it to be widely applicable to functions that traverse datastructures or other inductively defined data types (including the natural numbers). This includes functions like `map`, `filter`, `foldr`, `member`, `union` of functional programming [15, 27] and it should be clear that the approach based on types generalises well to passing funtions with known termination behaviour around as first class citizens.

To assess the strength of the method presented here we note that it will be clear from our approach that it applies to any function that *strictly* observes the rules for being in *primitive recursive form*. As our main running example shows we can even step outside primitive recursive form and show that Ackerman's function always terminates.

Related work on termination analysis

Termination analysis is an abstraction of time-complexity in the sense that having a time-bound on a computation also ensures termination. Automatic analysis of time-complexity, say by guessing and solving recurrence equations [12, 23], is very hard and cannot always succeed due to the undecidability of the problem although an impressive study is contained in [9]. Similar remarks go for automatic termination analysis where it is more often a well-founded order [6] (for recursive calls, or iteration) that needs to be guessed and verified. However, one must be careful when reading the literature on automatic time-complexity: too often one aims at establishing bounds that are only partially correct in the sense that *if* the program terminates *then* it will at most have used the time stated [21, 22]. Such approaches are useless for automatic termination analysis.

Automatic techniques that always succeed for termination analysis (and time-complexity) must necessarily have some weaknesses; an example is [7] that essentially gives up on recursive programs (e.g. by producing a time-estimate of "infinity"). A better approach may be to adapt the linear restraints of [4]. Not surprisingly the strongest techniques to termination analysis have been developed in terms of logical approaches [17, 16, 8, 2, 28]

whose implementation then often requires an *oracle* to resolve the non-syntax directed parts of the analysis. Our goal is to obtain an automatic analysis, by means of an inference system, that is sound and able to handle a reasonably large class of mainly datastructure-traversing functions and that interacts well with strongly typed languages. To this end we study a typed functional language with algebraic data types and an eager semantics. We do not develop an algorithm for inferring termination types although we conjecture that this may indeed be possible as is briefly discussed in the Appendix.

Throughout we shall restrict our attention to eager languages, in fact an eager functional language with pattern matching, as laziness (of data type constructors) presents formidable complications (although attempted in [24, 3]). A problem related to the study of termination analysis is that of quasi-termination analysis [10] that is relevant for partial evaluation: that the program only goes through a finite number of different configurations. Also termination has been studied for term rewriting systems (e.g. [13]) and for logic languages (e.g. [1]) but our approach relies heavily upon the algebraic data types of our language and the nature of the operational semantics.

Main aims and overview

Our study has been motivated by an investigation into the semantic principles needed to show the correctness of non-trivial termination analyses. We have decided to take an operational approach, rather than a denotational approach, because a long term goal is to be able to handle multi-paradigmatic languages where even the construction of a denotational semantics is not a trivial task. To this end we identify a need, not present in denotational semantics, of extending the semantic universe with new constructs, merely in order to be able to conduct certain kinds of proofs. Having done this one needs to investigate to which extent this means redoing the analysis for the new constructs. This is a problem that does not arise in denotational semantics and seems to be related to the full abstractness problem for denotational semantics: this is not of relevance for operational semantis but our works seems to indicate that related problems crop up anyway in order to facilitate conducting non-trivial proofs.

Section 2 defines the language, a fragment of Standard ML [15], that we will be working with: it has algebraic data types and uses pattern matching in preference to the general conditional. In addition we define the operational semantics [20] in form of a natural-style (or big-step) semantics and this is mostly straightforward. To prepare for the soundness proof we must extend the "semantic universe" with new primitives: functions $\texttt{FUN}_k$ that limit the number of recursive unfoldings to k (useful for the analogue of fixed point induction); and functions $\texttt{FUN}[\prec, \vec{w}, \overline{w}]$ for enforcing termination upon arguments not dominated by the parameter list (useful for establishing termination information).

In Section 3 we then develop the termination analysis. Following recent trends in program analysis we shall specify termination analysis by means of an inference system [11, 29] and annotate the type constructors. For the analysis of functions we ensure that the order of parameters in not important in that *all permutations* of parameters will be considered (before concluding that a function is not total).

Section 4 is devoted to showing soundness by proving results corresponding to monotonic-

ity, continuity and inclusiveness (of predicates) that would be expected in a denotational approach; here the $\texttt{FUN}[\prec, \vec{w}, \overline{w}]$ functions present complications as they are not monotonic. Similar complications are to be expected even in denotational semantics since one cannot prove functions total (i.e. terminating) using inclusive predicates.

In Section 5 the concluding remarks focus on the lesson learned about how to structure proofs of analyses when based on operational semantics. In the Appendix we provide the detailed proofs (although the highlights are usually presented in the main text) and we briefly discuss how to modify the usual algorithm for type inference [14] so as to obtain our totality types.

2 Syntax and Semantics

All programs may rely on the existence of the booleans as could have been introduced by the algebraic type definition

```
DEF 'Bool = True + False
```

As in Miranda [27] we write constructors with an initial capital letter and the Standard ML [15] convention of using quotes to indicate type variables is "generalized" to apply also to type constants. The definition of Ackerman's function then is

```
DEF 'Num = Zero + Succ 'Num
FUN ack Zero m => Succ m
  & ack (Succ n) Zero => ack n (Succ Zero)
  & ack (Succ n) (Succ m) => ack n (ack (Succ n) m)
ack
```

where the last line is the "program" to be executed (a call to Ackerman's function defined immediately above).

Here we introduce the natural numbers as an algebraic data type and then use pattern matching to define Ackerman's function: `ack 0 m = m + 1`, `ack (n + 1) 0 = ack n 1` and `ack (n + 1) (m + 1) = ack n (ack (n + 1) m)`. We shall not follow Miranda in allowing to write `(n + 1)` for the pattern `(Succ n)` as this would only complicate the technical development without adding new insights. But we should point out that it is a deliberate, and to some extent crucial, choice to use pattern matching rather than conditional to select among the three clauses of the definition.

The semantics will make clear that the language is *eager* (or strict) and so is closer to Standard ML than Miranda; in particular this means that data structures are finite and therefore no spurious numbers (the "infinite ones") are included in `'Num` (as would have been the case in Miranda). Hence Ackerman's function as defined here will be a total function and our analysis will be able to show this. We will obtain this information by inferring the annotated type

$$!^{\mathtt{T}}\ \mathtt{'Num} \underset{\mathtt{T}}{\rightarrow} \mathtt{'Num} \underset{\mathtt{T}}{\rightarrow} \mathtt{'Num}$$

$$
\begin{aligned}
&prog \ ::= defn\ block\ e\\
&defn \ ::= \epsilon \mid defn\ defn\\
&\qquad \mid \texttt{DEF}\ tv_1 \ldots tv_{p\geq 0}\ tc\ = c_1\ t_{11} \ldots t_{1(m_1\geq 0)} + \ldots + c_{n\geq 1}\ t_{n1} \ldots t_{n(m_n \geq 0)}\\
&block \ ::= \epsilon \mid block\ block\\
&\qquad \mid \texttt{FUN}\ v\ p_{11} \ldots p_{1(m\geq 1)} \Rightarrow e_1\ \&\ \ldots\ \&\ v\ p_{(n\geq 1)1} \ldots p_{nm} \Rightarrow e_n\\
&e \ ::= v \mid c \mid e\ e\ \mid \texttt{IF}\ e\ \texttt{THEN}\ e\ \texttt{ELSE}\ e\\
&v \ ::= \texttt{x} \mid \texttt{y} \mid \texttt{z} \mid \ldots\\
&p \ ::= v \mid c\ p_1 \ldots p_{n\geq 0}\\
&c \ ::= \texttt{Nil}\ \mid \texttt{Cons}\ \mid \texttt{Zero} \mid \ldots\\
&t \ ::= t_1 \ldots t_{n\geq 0}\ tc \mid t \underset{a}{\rightarrow} t \mid tv\\
&tc \ ::= \texttt{'List}\ \mid \texttt{'Int}\ \mid \ldots\\
&tv \ ::= \texttt{'a} \mid \texttt{'b} \mid \ldots \mid \texttt{''a}\ \mid \texttt{''b}\ \mid \ldots\\
&a \ ::= av \mid \texttt{T} \mid \epsilon\\
&av \ ::= \texttt{'1}\ \mid \texttt{'2}\ \mid \ldots\\
&ts \ ::= t\ \mid \forall\, tv.\ ts \mid \forall\, av.\ ts
\end{aligned}
$$

Figure 1: Abstract Syntax

The first $!^{\texttt{T}}$ indicates that the program does in fact terminate and produces a function `ack`. The first subscript `T` says that supplying an argument `n` to `ack` still terminates giving a function `ack n`. The final subscript `T` is non-trivial and says that giving the function yet another argument `m` the computation still terminates giving the result `ack n m`. In this notation `T` indicates a total function whereas an ϵ (to be thought of as the empty symbol and hence "invisible") indicate a possibly partial function; so our notation is a conservative extension of the usual one.

$$
[prog] \quad \frac{[\,] \vdash block\ \Downarrow \rho \qquad \rho \vdash e \Downarrow w}{\vdash defn\ block\ e\ \Downarrow w}
$$

$$
[\epsilon] \quad \rho \vdash \epsilon \Downarrow \rho
$$

$$
[;] \quad \frac{\rho \vdash block_1 \Downarrow \rho_1 \qquad \rho_1 \vdash block_2 \Downarrow \rho_2}{\rho \vdash block_1\ block_2 \Downarrow \rho_2}
$$

$$
[FUN] \quad \rho \vdash (\texttt{FUN}\ v\ \vec{p_1} \Rightarrow e_1 \ldots) \Downarrow \rho[v \mapsto \texttt{FUN}\ \rho\ v\ \vec{p_1} \Rightarrow e_1 \ldots]
$$

Figure 2: Semantics of Programs and Blocks

Syntax

The abstract syntax is summarized in Figure 1. A program ($prog \in \mathbf{Prog}$) consists of a definition ($defn \in \mathbf{Defn}$) of a series of algebraic data types followed by a block ($block \in \mathbf{Block}$) introducing a series of recursive functions and then an expression ($e \in \mathbf{Exp}$). The algebraic data types are *not* intended to be mutually recursive and at some point in the development we shall make the simplifying assumption that function types cannot be components of algebraic data types. The recursive functions are not intended to be mutually recursive either but allow pattern matching in several levels simultaneously; part of the well-formedness condition to be formulated later is that the patterns must be exhaustive.

The syntax of expressions, variables ($v \in \mathbf{Var}$), patterns ($p \in \mathbf{Pat}$) and constructors ($c \in \mathbf{Con}$) present no surprises. A small syntactic convenience is that we allow to write e.g. $tv_{p\geq 0}$ for tv_p with a side condition $p \geq 0$. In the course of the development we shall see that the syntax for expressions will have to be extended and later we shall argue in favour of this piecemeal approach as opposed to introducing all auxiliary syntactic constructs right at the start.

Types ($t \in \mathbf{Type}$) include parameterized type constants ($tc \in \mathbf{TyCon}$) and type variables ($tv \in \mathbf{TyVar}$) as well as function types. The syntax for function types allows labelling the arrow with an *annotation* ($a \in \mathbf{Ann}$). Possibilities include $a = \epsilon$ for arbitrary (possibly partial) functions and $a = \mathtt{T}$ for functions guaranteed to be total; the final possibility of annotation variables ($av \in \mathbf{AnnVar}$) is analogous to type variables and will be clarified in the type inference system. Type variables are of two kinds: singly quoted ones that may be instantiated to arbitrary types, and doubly quoted ones that may be instantiated only to types not containing function types (or singly quoted type variables).

Type schemes ($ts \in \mathbf{TyScheme}$) are types quantified over by type variables or annotation variables; we shall sometime write $tav \in \mathbf{TyVar} \cup \mathbf{AnnVar}$ to reduce the number of cases to be considered. A *closed* type, or monotype, is one without type or annotation variables whereas a polytype is a type that may well contain type or annotation variables. In a similar way a closed type scheme is one where all free type and annotation variables of the underlying type have been universally quantified.

Semantics

The operational semantics is of the big-step variety (also called natural semantics) where a syntactic construct evaluates to the required value in one big step. The semantics of programs and blocks is given in Figure 2. The purpose of the elaboration of a block is to extend a given environment ($\rho \in \mathbf{Env}$) to an extended environment incorporating the function definitions of the block. We shall regard an environment as a list of pairs of variables and values but written in a more readable syntax and relying on the usual convention of locating values in the environment by using the value of the rightmost pair whose variable is the one looked for. For functions the value bound into the environment is the function abstraction itself but extended with the environment at the point of declaration so as to obtain static scope. We do not need semantic rules for elaborating definitions; in our approach this will be a task for the type inference system.

$$v : w \rightsquigarrow [v \mapsto w]$$

$$\frac{p_i : w_i \rightsquigarrow \rho_i \quad (\text{FORALL } i = 1, \ldots, (m \geq 0))}{(p_1, \ldots, p_m) : (w_1, \ldots, w_m) \rightsquigarrow [\,]\rho_1 \ldots \rho_m}$$

$$\frac{p_i : w_i \rightsquigarrow \rho_i \quad (\text{FORALL } i = 1, \ldots, (m \geq 0))}{c\ p_1 \ldots p_m : c\ w_1 \ldots w_m \rightsquigarrow [\,]\rho_1 \ldots \rho_m}$$

$$\frac{p_i : w_i \not\rightsquigarrow}{(p_1, \ldots, p_m) : (w_1, \ldots, w_m) \not\rightsquigarrow}$$

$$(p_1, \ldots, p_m) : (w_1, \ldots, w_{m'}) \not\rightsquigarrow \quad \text{IF } m \neq m'$$

$$\frac{p_i : w_i \not\rightsquigarrow}{c\ p_1 \ldots p_m : c\ w_1 \ldots w_m \not\rightsquigarrow}$$

$$c\ p_1 \ldots p_m : c' w_1 \ldots w_{m'} \not\rightsquigarrow \qquad \text{IF } m \neq m' \vee\ c \neq c'$$

Figure 3: Semantics of Matching

To handle the semantics of function application we need to be able to match a pattern against a value: if it succeeds we get a new small environment and write $p : w \rightsquigarrow \rho$; if it fails we write $p : w \not\rightsquigarrow$. More generally we may match a tuple of patterns against a tuple of values and this is achieved by the rules and axioms given in Figure 3. We have used the notation $p_i : w_i \rightsquigarrow \rho_i$ (FORALL $i = 1, \ldots, m \geq 0$) as a shorthand for the more usual $p_1 : w_1 \rightsquigarrow \rho_1 \ldots p_m : w_m \rightsquigarrow \rho_m$ together with the side condition $m \geq 0$. In this way the rules and axioms for composite patterns are applicable also for constructors: so $c : c \rightsquigarrow [\,]$ where $[\,]$ denotes the empty environment and $c : c' \not\rightsquigarrow$ when $c \neq c'$.

The eager semantics of expressions is given in Figure 4. The general form of a judgement is $\rho \vdash e \Downarrow w$ and says that in the environment ρ the expression e evaluates to the value w. The values ($w \in \mathbf{Val}$) are given by

$$w \ ::= (\texttt{FUN}\ \rho\ v\ \vec{p_1} \Rightarrow \overline{e}_1 \ldots)\ w_1 \ldots w_{m<|\vec{p_1}|} \ \mid\ c\ w_1 \ldots w_{m \geq 0}$$

and those constructed by the semantics will not contain any free variables; we shall write this as $FV(w) = \emptyset$ for a hopefully obvious definition of FV.

The first few axioms and rules are straightforward. Rules $[APP^{<}_{FUN}]$ and $[APP^{=}_{FUN}]$ both deal with the application of a function requiring $\mid \vec{p_1} \mid$ arguments, where $\mid \vec{p_1} \mid$ is the number of patterns in the list, to m arguments. The first rule considers the case $m < \mid \vec{p_1} \mid$ where the function remains less than fully applied. The second rule deals with the case $m = \mid \vec{p_1} \mid$ where the function becomes fully applied and hence has to be "unfolded". We take care to change to the definition-time environment and to extend it with the function applied; this gives static scope and allows recursive calls. Additionally, we must select the right branch of the function body and extend the environment with a binding of formal parameter variables to actual argument values. This could have been

$[v]$ $$\rho \vdash v \Downarrow \rho(v)$$

$[c]$ $$\rho \vdash c \Downarrow c$$

$[IF_T]$ $$\frac{\begin{array}{c}\rho \vdash e_1 \Downarrow \texttt{True} \\ \rho \vdash e_2 \Downarrow w\end{array}}{\rho \vdash \texttt{IF}\ e_1\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3 \Downarrow w}$$

$[IF_F]$ $$\frac{\begin{array}{c}\rho \vdash e_1 \Downarrow \texttt{False} \\ \rho \vdash e_3 \Downarrow w\end{array}}{\rho \vdash \texttt{IF}\ e_1\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3 \Downarrow w}$$

$[APP_C]$ $$\frac{\begin{array}{l}\rho \vdash e_1 \Downarrow c\ w_1 \ldots w_{m-1} \\ \rho \vdash e_2 \Downarrow w_m\end{array}}{\rho \vdash e_1\ e_2 \Downarrow c\ w_1 \ldots w_m}$$

$[APP^{<}_{FUN}]$ $$\frac{\begin{array}{l}\rho \vdash e_1 \Downarrow (\texttt{FUN}\ \overline{\rho}\ v\ \vec{p_1} \Rightarrow \overline{e}_1 \ldots)w_1 \ldots w_{m-1} \\ \rho \vdash e_2 \Downarrow w_m\end{array}}{\rho \vdash e_1\ e_2 \Downarrow (\texttt{FUN}\ \overline{\rho}\ v\ \vec{p_1} \Rightarrow \overline{e}_1 \ldots)w_1 \ldots w_m}$$

IF $m <\mid \vec{p_1} \mid$

$[APP^{=}_{FUN}]$ $$\frac{\begin{array}{l}\rho \vdash e_1 \Downarrow (\texttt{FUN}\ \overline{\rho}\ v\ \vec{p_1} \Rightarrow \overline{e}_1 \ldots)w_1 \ldots w_{m-1} \\ \rho \vdash e_2 \Downarrow w_m \\ \overline{\rho}[v \mapsto \texttt{FUN}\ \overline{\rho}\ v\ \vec{p_1} \Rightarrow \overline{e}_1 \ldots] \vdash (\lambda \vec{p_1}.\overline{e}_1 \ldots)w_1 \ldots w_m \Downarrow w\end{array}}{\rho \vdash e_1\ e_2 \Downarrow w}$$

IF $m =\mid \vec{p_1} \mid$

$[\lambda_1]$ $$\frac{\begin{array}{l}(p_{11}, \ldots, p_{1m}) : (w_1, \ldots, w_m) \leadsto \rho' \\ \rho, \rho' \vdash \overline{e}_1 \Downarrow w\end{array}}{\rho \vdash (\lambda p_{11} \ldots p_{1m}.\overline{e}_1 \ldots)w_1 \ldots w_m \Downarrow w}$$

$[\lambda_2]$ $$\frac{\begin{array}{c}(p_{11}, \ldots, p_{1m}) : (w_1, \ldots, w_m) \not\leadsto \\ \rho \vdash (\lambda p_{21} \ldots p_{2m}.\overline{e}_2 \ldots)w_1 \ldots w_m \Downarrow w\end{array}}{\rho \vdash (\lambda p_{11} \ldots p_{1m}.\overline{e}_1\ \&\ p_{21} \ldots p_{2m}.\overline{e}_2 \ldots)w_1 \ldots w_m \Downarrow w}$$

$[w]$ $$\rho \vdash \texttt{FUN}\ \overline{\rho}\ v\ \vec{p_1} \Rightarrow \overline{e}_1 \ldots \Downarrow \texttt{FUN}\ \overline{\rho}\ v\ \vec{p_1} \Rightarrow \overline{e}_1 \ldots$$

Figure 4: Semantics of Expressions and Temporaries

incorporated into rule $[APP^{=}_{FUN}]$ but to avoid an overly complicated rule we have added the two rules $[\lambda_1]$ and $[\lambda_2]$ for selecting and evaluating the appropriate branch. To this end we have introduced temporary expressions (e.g. $te \in$ **TempExp**) given by

$$te ::= (\lambda\ p_{11} \ldots p_{1(m\geq 1)}.e_1\ \&\ \ldots \&\ p_{(n\geq 0)1} \ldots p_{nm}.e_n)\ w_1 \ldots w_m$$

Since temporary expressions cannot be written in a program and cannot be bound into

environments or values we do *not* include temporary expressions in the syntax of expressions or values. Also note that the use of pattern matching saves us the trouble of adding explicit destructor functions.

A small final point is that we shall find it helpful to regard values as a special kind of expressions. This may be achieved by extending the syntax of expressions by[1]

$$e ::= \ldots \mid \texttt{FUN}\ \rho\ v\ p_{11} \ldots p_{1(m \geq 1)} \Rightarrow e_1\ \&\ \ldots \&\ v\ p_{(n \geq 1)1} \ldots p_{nm} \Rightarrow e_n$$

and by having the axiom $[w]$ in Figure 4.

Example 1 Let *defn* and *block* be the definitions of `Num` and `ack` in Ackerman's function, respectively. Then one may verify

$$\vdash \mathit{defn}\ \mathit{block}\ \ \texttt{ack}\ (\texttt{Succ}^2\ \texttt{Zero})(\texttt{Succ}^3\ \texttt{Zero}) \Downarrow (\texttt{Succ}^9\ \texttt{Zero})$$

where ($\texttt{Succ}^2$ `Zero`) abbreviates `Succ` (`Succ` `Zero`) etc. □

Extending the semantic universe

In denotational semantics the semantic domains may contain elements that are not denotable as the semantics of programs or other syntactic constructs. This phenomenon lies at the heart of the full abstractness problem for denotational semantics but it also facilitates a number of proofs where such elements can be used. In operational semantics the semantic universe is no larger than what has been introduced: either in the syntax or as new auxiliary forms (e.g. `FUN` $\rho\ v\ \ldots$) facilitating the definition of the semantics. Thus when performing certain proofs we shall find a need to extend the semantic universe with new auxiliary forms whose *sole purpose* is to facilitate a certain method of proof when establishing a theorem. We shall see this need arise in the soundness proof and to avoid cluttering the paper we present the extensions here; however, it may be advisable to postpone the reading until needed for the soundness proof. In the conlusion we will then comment on the overall picture that we see emerging in proofs based on operational semantics.

The first extension is a syntactic device

$$\texttt{FUN}_k\ \overline{\rho}\ v\ \vec{p_1} \Rightarrow \overline{e}_1\ \&\ \ldots \&\ v\ \vec{p_n} \Rightarrow \overline{e}_n$$

that allows only k unfoldings of the function v. Formally we extend the syntax of values (and similarly expressions) by

$$w ::= \ldots \mid (\texttt{FUN}_k\ \overline{\rho}\ v\ \vec{p_1} \Rightarrow \overline{e}_1 \ldots)\ w_1 \ldots w_{m < |\vec{p_1}|}$$

and the semantics by the following modifications of the rules of Figure 4:

[1] Since we use a big-step semantics there is no harmful ambiguity between viewing a syntactic construct as a value or as an expression (when both are possible). Hence we do not follow [19] in enclosing values in angle brackets.

$[w]'$ $\qquad \rho \vdash \mathtt{FUN}_k\ \overline{\rho}\ v\ \vec{p_1} \Rightarrow \overline{e}_1 \ldots \Downarrow \mathtt{FUN}_k\ \overline{\rho}\ v\ \vec{p_1} \Rightarrow \overline{e}_1 \ldots$

$$[APP^{<}_{FUN}]' \qquad \frac{\rho \vdash e_1 \Downarrow (\mathtt{FUN}_k\ \overline{\rho}\ v\ \vec{p_1} \Rightarrow \overline{e}_1 \ldots)\ w_1 \ldots w_{m-1} \quad \rho \vdash e_2 \Downarrow w_{m<|\vec{p_1}|}}{\rho \vdash e_1\ e_2 \Downarrow (\mathtt{FUN}_k\ \overline{\rho}\ v\ \vec{p_1} \Rightarrow \overline{e}_1 \ldots)\ w_1 \ldots w_{m-1}\ w_m}$$

$$[APP^{=}_{FUN}]' \qquad \frac{\rho \vdash e_1 \Downarrow (\mathtt{FUN}_{k+1}\ \overline{\rho}\ v\ \vec{p_1} \Rightarrow \overline{e}_1 \ldots)\ w_1 \ldots w_{m-1} \quad \rho \vdash e_2 \Downarrow w_{m=|\vec{p_1}|} \quad \overline{\rho}\ [v \mapsto \mathtt{FUN}_k\ \overline{\rho}\ v\ \vec{p_1} \Rightarrow \overline{e}_1 \ldots] \vdash (\lambda\ \vec{p_1}.\overline{e}_1 \ldots)\ w_1 \ldots w_{m-1}\ w_m \Downarrow w}{\rho \vdash e_1\ e_2 \Downarrow w}$$

Thus in the clause for a fully applied function we allow $\mathtt{FUN}_{k+1}$ to unfold with access to $\mathtt{FUN}_k$ and $\mathtt{FUN}_0$ is not allowed to unfold at all.

The second extension is a syntactic device

$$\mathtt{FUN}[\prec, \overline{w}_1, \ldots, \overline{w}_m, \overline{w}]\overline{\rho}\ v\ \vec{p_1} \Rightarrow \overline{e}_1\ \&\ \ldots \&\ v\ \vec{p_n} \Rightarrow \overline{e}_n$$

that allows to modify a function to arbitrarily produce the result $\overline{w}$ upon arguments $w_1, \ldots, w_m$ that do not satisfy $(w_1, \ldots, w_m) \prec (\overline{w}_1, \ldots, \overline{w}_m)$ for some relation $\prec$. Formally we extend the syntax of values (and similarly expressions) by

$$w ::= \ldots \mid (\mathtt{FUN}[\prec, \overline{w}_1, \ldots, \overline{w}_{|\vec{p_1}|}, \overline{w}]\ \overline{\rho}\ v\ \vec{p_1} \Rightarrow \overline{e}_1 \ldots)\ w_1 \ldots w_{m<|\vec{p_1}|}$$

and the semantics by the following modification of the rules of Figure 4:

$[w]''$ $\qquad \rho \vdash (\mathtt{FUN}[\prec, \overline{w}_1, \ldots, \overline{w}_m, \overline{w}]\ \overline{\rho}\ v\ \vec{p_1} \Rightarrow \overline{e}_1 \ldots) \Downarrow (\mathtt{FUN}[\prec, \overline{w}_1, \ldots \overline{w}_m, \overline{w}]\ \overline{\rho}\ v\ \vec{p_1} \Rightarrow \overline{e}_1 \ldots)$

$$[APP^{<}_{FUN}]'' \qquad \frac{\rho \vdash e_1 \Downarrow (\mathtt{FUN}[\prec, \overline{w}_1, \ldots, \overline{w}_{|\vec{p_1}|}, \overline{w}]\ \overline{\rho}\ v\ \vec{p_1} \Rightarrow \overline{e}_1 \ldots)\ w_1 \ldots w_{m-1} \quad \rho \vdash e_2 \Downarrow w_{m<|\vec{p_1}|}}{\rho \vdash e_1\ e_2 \Downarrow (\mathtt{FUN}[\prec, \overline{w}_1, \ldots, \overline{w}_{|\vec{p_1}|}, \overline{w}]\ \overline{\rho}\ v\ \vec{p_1} \Rightarrow \overline{e}_1 \ldots)\ w_1 \ldots w_{m-1}\ w_m}$$

$$[APP^{=}_{FUN}]''_1 \qquad \frac{\rho \vdash e_1 \Downarrow (\mathtt{FUN}[\prec, \overline{w}_1, \ldots, \overline{w}_m, \overline{w}]\ \overline{\rho}\ v\ \vec{p_1} \Rightarrow \overline{e}_1 \ldots)\ w_1 \ldots w_{m-1} \quad \rho \vdash e_2 \Downarrow w_{m=|\vec{p_1}|} \quad \overline{\rho}[v \mapsto \mathtt{FUN}\ \overline{\rho}\ v\ \vec{p_1} \Rightarrow \overline{e}_1 \ldots] \vdash (\lambda\ \vec{p_1}.\overline{e}_1 \ldots)\ w_1 \ldots w_{m-1}\ w_m \Downarrow w \quad (w_1, \ldots, w_m) \prec (\overline{w}_1, \ldots, \overline{w}_m)}{\rho \vdash e_1\ e_2 \Downarrow w}$$

$$[APP^{=}_{FUN}]''_2 \qquad \frac{\rho \vdash e_1 \Downarrow (\mathtt{FUN}[\prec, \overline{w}_1, \ldots, \overline{w}_m, \overline{w}]\ \overline{\rho}\ v\ \vec{p_1} \Rightarrow \overline{e}_1 \ldots)\ w_1 \ldots w_{m-1} \quad \rho \vdash e_2 \Downarrow w_{m=|\vec{p_1}|} \quad \neg((w_1, \ldots, w_m) \prec (\overline{w}_1, \ldots, \overline{w}_m))}{\rho \vdash e_1\ e_2 \Downarrow \overline{w}}$$

The latter two rules show the different behaviour of $\mathtt{FUN}[\prec, \overline{w}_1, \ldots, \overline{w}_m, \overline{w}] \ldots$ upon arguments depending on their relation to $(\overline{w}_1, \ldots, \overline{w}_m)$. Note that in $[APP^{=}_{FUN}]''_1$ it is FUN, not $\mathtt{FUN}[\prec, \overline{w}_1, \ldots, \overline{w}_m, \overline{w}]$, that is made available to the recursive calls. It is when constructing these expressions that it is important that all algebraic data types do contain *at least one element* that can be used for $\overline{w}$.

We shall call $\mathtt{FUN}_k$ for a *constrained* version of FUN and $\mathtt{FUN}[\prec, \overline{w}_1, \ldots, \overline{w}_m, \overline{w}]$ for a *forced* version of FUN. When referring to the rules of the semantics we shall allow dispensing with the superscript dashes and the subscript integer as this is not likely to cause any confusion. We shall say that an entity (e.g. an expression or an environment) is *pure* if it contains no forced version of FUN whereas constrained versions are permitted. The motivation behind this notion is that some of our subsequent results (e.g. Lemmas 12 and 13) are not valid for forced versions of FUN. However, it does simplify our task (mainly in the proof of Lemma 18) to be able to use forced versions when due care is taken.

3 Termination Inference

When analysing a recursive function the idea will be to ensure termination by comparing the *formal* parameters (as given by the patterns of the function definition) with the *actual* parameters of the recursive calls in the body of the recursive function definition. To facilitate this the analysis of expressions will not only determine the type of the expression but also a set (W below) of recursive calls. To control the size of this set we also maintain a set of relevant function names (*cenv* below).

Following recent trends in program analysis we shall specify the termination analysis by means of an inference system (as opposed to an abstract interpretation). Since the types play an important role for our analysis the inference system takes the form of an extended inference system for type analysis.

Before giving the details of this inference system we give a brief overview of the different kinds of judgement involved in the system. The judgement for *programs* is

$$\vdash \mathit{defn}\ \mathit{block}\ e : !^a\ t$$

where a and t are (not necessarily closed) annotations and types, respectively. The basic idea is that the program must terminate if $a = \mathtt{T}$ and that if it terminates the result will be as described by the annotated type t. The judgement for *definitions* is of the form

$$\mathit{denv}_1 \vdash \mathit{defn} \Rightarrow \mathit{denv}_2$$

and extends the existing definition environment denv_1 with the local definitions so as to produce the new definition environment denv_2; this involves recording the arity of each type constructor and the type scheme of each constructor. The judgement for *blocks* is of the form

$$\mathit{denv}, \mathit{tenv}_1 \vdash \mathit{block} \Rightarrow \mathit{tenv}_2$$

and extends the existing type environment $tenv_1$ with the local functions declarations so as to produce the new type environment $tenv_2$; this involves recording the type scheme of each function declared. Finally, the judgement for *expressions* is of the form

$$denv,\ tenv,\ cenv \vdash e : !^a\ t\ \&\ W$$

Here the idea is that W is a set of those *maximal syntactic* constructs $v\ e'_1 \ldots e'_k$ that may be found inside the expression e where v is one of the functions selected for "monitoring" in the "calling environment" $cenv$. The environments $denv$ and $tenv$ are as above and the annotation a and the type t are as for programs.

Programs and definitions

We are now ready to provide the details of the annotated type system. As said above the judgement for programs is $\vdash defn\ block\ e : !^a\ t$ where a and t are (not necessarily closed) annotations and types, respectively. The intent is that the program must terminate if $a = \mathtt{T}$ and that if it terminates the result will be as described by the annotated type t.

$$[prog]\quad \frac{\begin{array}{l}[\mathtt{'Bool}\ :0][\mathtt{True}\ \mapsto \mathtt{'Bool}\ ][\mathtt{False}\ \mapsto \mathtt{'Bool}\ ] \vdash defn \Rightarrow denv \\ denv, [\,] \vdash block \Rightarrow tenv \\ denv, tenv, \emptyset \vdash e : !^a\ t\ \&\ W\end{array}}{\vdash defn\ block\ e : !^a\ t}$$

$$[\epsilon]\quad denv \vdash \epsilon \Rightarrow denv$$

$$[;]\quad \frac{\begin{array}{c}denv \vdash defn_1 \Rightarrow denv_1 \\ denv_1 \vdash defn_2 \Rightarrow denv_2\end{array}}{denv \vdash defn_1\ defn_2 \Rightarrow denv_2}$$

$$[DEF]\quad \frac{\begin{array}{l}denv[tc : p] \vdash t_{ij} : true\ (\text{FORALL } i = 1, \ldots, n)\ (\text{FORALL } j = 1, \ldots, m_i) \\ denv[tc : p] \vdash tv_i : true\ (\text{FORALL } i = 1, \ldots, p) \\ \{tc\} \cap \text{dom}\ (denv) = \emptyset \\ \text{FTV}\ (t_{ij}) \subseteq \{tv_1, \ldots, tv_p\} \wedge\ \text{FAV}\ (t_{ij}) = \emptyset\ \ (\text{FORALL } i)\ \ (\text{FORALL } j) \\ \exists i : denv \vdash t_{ij} : true\ \ (\text{FORALL } j = 1, \ldots, m_i) \\ \{c_1, \ldots, c_n\} \cap \text{dom}\ (denv) = \emptyset\end{array}}{\begin{array}{r}denv \vdash \mathtt{DEF}\ tv_1 \ldots tv_p\ tc = c_1\ t_{11} \ldots t_{1m_1} + \ldots + c_n\ t_{n1} \ldots t_{nm_n} \Rightarrow \\ denv[tc : p] \ldots [c_i : \forall tv_1 \ldots \forall tv_p . t_{i1} \underset{\mathtt{T}}{\rightarrow} \ldots \underset{\mathtt{T}}{\rightarrow}\ t_{im_i} \underset{\mathtt{T}}{\rightarrow} tv_1 \ldots tv_p\ tc] \ldots\end{array}}$$

Figure 5: Analysis of Programs and Definitions

There is just one rule applicable to programs; it is listed in Figure 5 as rule [*prog*]. The first step is to process the definitions and to obtain a definition environment. An example definition environment is

$$[\mathtt{'Bool}\ :0][\mathtt{True}\ \mapsto \mathtt{'Bool}\ ][\mathtt{False}\ \mapsto \mathtt{'Bool}\ ]$$

that records a type constructor `'Bool` of arity 0 and two constructors `True` and `False` of the expected type.

Turning to definitions we recall the form, $denv_1 \vdash defn \Rightarrow denv_2$, of the judgements. Three axioms and rules ared defined in Figure 5. The axiom $[\epsilon]$ and the rule $[;]$ express that the definition environment is obtained by traversing the definition and extending the definition environment along the way.

The rule $[DEF]$ records the effect of an algebraic data type definition: the type constructor and its arity must be recorded and for each constructor we record the appropriate type scheme. We demand that a type constructor is never redefined (line 3 of the premiss) and also that the constructors are never redefined (line 6). For the development of the next section it is important that the algebraic data type does not contain function types (line 1) and that the type variables used as parameters will never be instantiated to function types either (essentially line 2). The details of these formulations are given in Figure 6: a type t is well-formed if $denv \vdash t : s$ for some s and if additionally $s = true$ it is *simple*, i.e. obeys the restriction of no function types. Returning to the rule $[DEF]$ we ensure that the type schemes recorded for the constructors are indeed *closed* by means of an explicit and rather natural condition (line 4). Finally, we want to avoid creating empty types like `DEF 'Empty = Empty 'Empty` and the formulation chosen (line 5) avoids the creation of empty types.

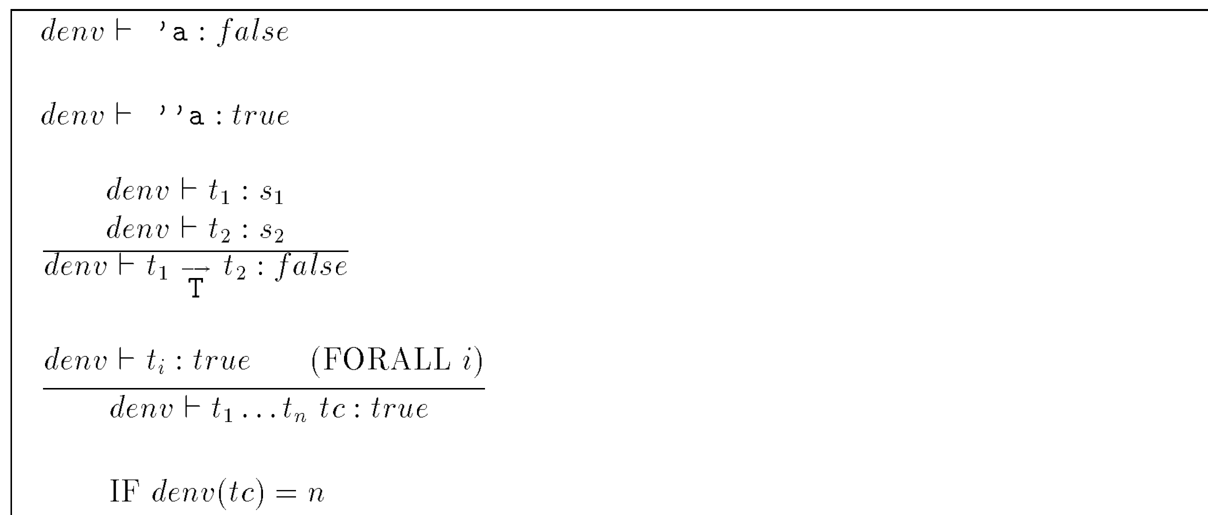

Figure 6: Type Well-formedness and Simplicity

Example 2 Having analysed the definition part of Ackerman's function the definition environment is $[$'Bool $= 0$, 'Num $: 0$, True $\mapsto$ Bool, False $\mapsto$ Bool, Zero $\mapsto$ 'Num, Succ $\mapsto$ 'Num $\xrightarrow[T]{}$ 'Num$]$. □

Expressions

We shall defer the treatment of blocks until we have explained the treatment of expressions. The judgement for expressions is $denv,\ tenv,\ cenv \vdash e : !^a\ t\ \&\ W$. Here $denv$ is a definition environment, i.e. a list of type constructors and their arity and a list of constructors and their closed type schemes. In a similar way $tenv$ is a type environment,

i.e. a list of variables and their type schemes; these type schemes need not be closed and may degenerate to ordinary types (which need not be closed either). The set *cenv* is a "calling environment" that indicates those recursive calls we wish to collect in the W component. The idea is that W is a set of those *maximal* constructs $v\ e'_1 \ldots e'_k$ that may be found inside the expression e; so whenever function v occurs in a context we attempt to determine the maximal number of arguments $e'_1 \ldots e'_k$ to v and then collect $v\ e'_1 \ldots e'_k$ (rather than $v\ e'_1 \ldots e'_{k'}$ for some $k' < k$) in W. The annotation a and type t are as for programs.

The formal definition is given in Figure 7. The axioms for variables collect the call of a single variable if that variable is in the calling environment; for constants we collect nothing. Since we are in an eager language variables and constants always evaluate, hence the $!^{\mathsf{T}}$ annotation. The type must be a generic instance of the appropriate type scheme and this is written

$$denv \vdash ts \geq t$$

where ts is the type scheme $denv(v)$ or $denv(c)$. To formalize this let a *substitution* be a finite mapping from type variables to types (not type schemes) and from annotation variables to annotations. It is *ground* if all types and annotations in its range are closed. It *covers* a given syntactic construct if the domain of the substitution includes all type and annotation variables of the syntactic construct. It is ($denv$-)well-formed if it respects simplicity of types and does not introduce types that are not well-formed; for a substitution U this may be written

$$denv \vdash tv : true \wedge tv \in \text{dom}\ (U) \Rightarrow denv \vdash U(tv) : true$$
$$denv \vdash tv : false \wedge tv \in \text{dom}\ (U) \Rightarrow \exists s : denv \vdash U(tv) : s$$

The notion of generic instance may then be clarified by

$$denv \vdash \forall tav_1 \ldots \forall tav_m.t \geq U(t)$$
IF U is a $denv$-well-formed substitution with domain $\{tav_1, \ldots, tav_m\}$

Applying a substitution to a type or an annotation is a simple structural procedure but when applying it to a type scheme (and the substitution is not ground) care must be taken to rename the bound type and annotation variables so as to avoid conflict.

The rules for application and conditional assume that types and annotations match. The calls to be collected are the union of the calls of the subexpressions. An exception arises for application in the case where $e_1 \in W_1$: in this case e_1 is in itself a maximal call and we must collect $e_1\ e_2$ instead of e_1. With the rules of Figure 7 this means that we may treat `IF ...THEN` $v\ e_1\ e$ `ELSE` $v\ e_2\ e$ more precisely than (`IF ...THEN` $v\ e_1$ `ELSE` $v\ e_2$) e in terms of the maximality of the calls collected. It is possible to improve upon this by a more refined version of the rules in Figure 7: the set W must be split into one component for maximal inner calls and one component for maximal calls that are "exposed to the continuation".

$$[VAR] \quad denv,\ tenv,\ cenv \vdash v : !^{\mathtt{T}}\ t\ \&\ \{v\} \cap cenv$$
$$\text{IF } denv \vdash tenv(v) \geq t$$

$$[CON] \quad denv,\ tenv,\ cenv \vdash c : !^{\mathtt{T}}\ t\ \&\ \emptyset$$
$$\text{IF } denv \vdash denv(c) \geq t$$

$$[APP] \quad \frac{\begin{array}{l} denv,\ tenv,\ cenv \vdash e_1 : !^a\ t_1 \underset{a}{\rightarrow} t_2\ \&\ W_1 \\ denv,\ tenv,\ cenv \vdash e_2 : !^a\ t_1\ \&\ W_2 \end{array}}{denv,\ tenv,\ cenv \vdash e_1\ e_2 : !^a\ t_2\ \&\ W}$$
$$\text{WHERE } W = W_2 \cup (W_1 \backslash \{e_1\}) \cup \begin{cases} \{e_1\ e_2\} & \text{if } e_1 \in W_1 \\ \emptyset & \text{if } e_1 \notin W_1 \end{cases}$$

$$[IF] \quad \frac{\begin{array}{c} denv,\ tenv,\ cenv \vdash e_1 : !^a\ \texttt{'Bool}\ \&\ W_1 \\ denv,\ tenv,\ cenv \vdash e_2 : !^a\ t\ \&\ W_2 \\ denv,\ tenv,\ cenv \vdash e_3 : !^a\ t\ \&\ W_3 \end{array}}{denv,\ tenv,\ cenv \vdash \texttt{IF}\ e_1\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3 : !^a\ t\ \&\ W_1 \cup W_2 \cup W_3}$$

$$[SUB] \quad \frac{denv,\ tenv,\ cenv \vdash e : !^a\ t\ \&\ W}{denv,\ tenv,\ cenv \vdash e : !^{a'}\ t'\ \&\ W}$$
$$\text{IF } a \rightsquigarrow a', t \rightsquigarrow t'$$

Figure 7: Analysis of Expressions

The subsumption rule *[SUB]* may be used to facilitate the use of rules [*APP*] and [*IF*] in which we assumed that types and annotations match. The annotation may be left unchanged or a `T` may be changed to anything. This is formalized by just two axioms:

$$a \rightsquigarrow a \qquad\qquad \mathtt{T} \rightsquigarrow a$$

For types we allow to change annotations in top-level covariant positions only. This may be formalized by one axiom and one rule:

$$t \rightsquigarrow t$$

$$\frac{a \rightsquigarrow a' \qquad t_2 \rightsquigarrow t_2'}{t_1 \underset{a}{\rightarrow} t_2 \rightsquigarrow t_1 \underset{a'}{\rightarrow} t_2'}$$

By restricting ourselves to covariant positions we avoid the problems of contravariance and by restricting ourselves to top-level positions we avoid the need to regard a "doubly contravariant" position as a "covariant" one.

Remark We conjecture that if the subsumption rule is restricted to be applicable only to variables and constants then this corresponds to what would have been obtained without a subsumption rule and with type schemes

$$\forall av_1 \ldots av_m. \quad t_1 \underset{av_1}{\rightarrow} \ldots t_m \underset{av_m}{\rightarrow} t$$

(for "fresh" annotation variables $av_1, \ldots, av_m$) instead of the

$$t_1 \underset{\mathtt{T}}{\rightarrow} \ldots t_m \underset{\mathtt{T}}{\rightarrow} t$$

used in the rule $[DEF]$ (and similarly $[FUN]$). □

Example 3 When analysing the last clause in the definition of Ackerman's function we shall see that we have

$denv$ as given by Example 2
$tenv = [\ \mathtt{ack} : \mathtt{'Num} \underset{\mathtt{T}}{\rightarrow} \mathtt{'Num} \underset{\mathtt{T}}{\rightarrow} \mathtt{'Num}\ ][\ \mathtt{n} \mapsto \mathtt{'Num}\ ][\mathtt{m} \mapsto \mathtt{'Num}\ ]$
$cenv = \{\mathtt{ack}\}$

The analysis

$$denv,\ tenv,\ cenv \vdash \mathtt{ack\ n\ (ack\ (Succ\ n)\ m)} : !^a\ t\ \&\ W$$

then yields

$a = \mathtt{T}$
$t = \mathtt{'Num}$
$W = \{(\mathtt{ack\ n\ (ack\ (Succ\ n)\ m)}), (\mathtt{ack\ (Succ\ n)\ m})\}$

In this case rule $[SUB]$ was not needed. □

Blocks

This leaves us with the analysis of blocks. The axiom $[\epsilon]$ and the rule $[;]$ in Figure 8 are as for definitions. To avoid an overly complicated rule for function declarations we have a separate rule $[GEN]$ for generalization: it applies to type variables as well as annotation variables. We use $FTV(t)$ and $FTV(tenv)$ to denote the sets of free type variables and $FAV(a)$ and $FAV(tenv)$ to denote the sets of free annotation variables.

The function declaration itself is handled by rule $[FUN_\pi]$ and extends the given type environment with the type of the recursively defined function itself. This involves guessing the types $t_1, \ldots, t_n$ of the argument and the type t of the result. We must therefore check the well-formedness of the type t (line 1 of the premiss) and of the types $t_1, \ldots, t_n$ (line 2). At the same time we obtain the small environments $tenv_1, \ldots, tenv_n$ that give the types of the variables embedded in the patterns. The details of this are given by the inference system of Figure 9; once again the rule for constructors may have $m = 0$ and so yields $denv \vdash c : t \Rightarrow [\,]$ if the side condition is fulfilled. The main premiss of $[FUN_\pi]$ is then the validation that the respective bodies give the correct type (line 3): the given environment is extended with the type of the recursive functions and of the variables embedded in the patterns.

The π subscripting rule $[FUN_\pi]$ is a permutation over $1, \ldots, m$. To allow postulating that we define a *total function* we must verify that the recursive calls are only applied to smaller arguments. These recursive calls were collected in the W_i components. Since there are many arguments it is natural to use a lexicographic order. It is given by

$$(e'_1, \ldots, e'_k) <^\pi_a \ (p_1, \ldots, p_m)$$
$$\Updownarrow$$
$$\exists j : \{\pi(1), \ldots, \pi(j)\} \subseteq \{1, \ldots, k\} \wedge$$
$$\quad \forall i < j : e'_{\pi(i)} \leq_a \ p_{\pi(i)} \wedge$$
$$\quad e'_{\pi(j)} <_a \ p_{\pi(j)}$$

where $\leq_a$ (with irreflexive part $<_a$) is responsible for recognizing that the corresponding position has (strictly) decreased. The key axioms are

$$e <_a c \ldots e \ldots$$

$$e_1 <_\epsilon \ e_2$$

stating that for a total function we must eventually find a call to a pattern that was contained in a larger pattern in the original call, and that for a possibly partial function we do not require this. On top of this we need the axiom

$$e \leq_a \ e$$

and the rules

$$\frac{e_1 <_a e_2}{e_1 \leq_a \ e_2}$$

$$\frac{e_1 \leq_a e_2 \quad e_2 \leq_a e_3}{e_1 \leq_a \ e_3}$$

$$\frac{e_1 \leq_a e_2 \quad e_2 <_a e_3}{e_1 <_a e_3}$$

$$\frac{e_1 <_a e_2 \quad e_2 \leq_a e_3}{e_1 <_a \ e_3}$$

in order to axiomatize that $\leq_a$ is a partial order with $<_a$ its irreflexive part.

Example 4 Consider the Ackerman function and let π be the trivial permutation (i.e. $\pi = (1, 2)$). We have

$$(\texttt{n, Succ Zero}) <^\pi_{\texttt{T}} \ (\texttt{Succ n, Zero})$$

because $\texttt{n} <_{\texttt{T}} \texttt{Succ n}$; next we have

$$(\texttt{Succ n, m}) \ <^\pi_{\texttt{T}} (\texttt{Succ n, Succ m})$$

because $\texttt{Succ n} \leq_{\texttt{T}} \ \texttt{Succ n}$ and $\texttt{m} <_{\texttt{T}} \ \texttt{Succ m}$; finally we have

$$(\texttt{n, ack (Succ n) m}) <^\pi_{\texttt{T}} (\texttt{Succ n, Succ m})$$

$[\epsilon]$ $\quad denv,\ tenv \vdash \epsilon \Rightarrow tenv$

$$[;]\quad \frac{\begin{array}{c} denv,\ tenv \vdash block_1 \Rightarrow tenv_1 \\ denv,\ tenv_1 \vdash block_2 \Rightarrow tenv_2 \end{array}}{denv,\ tenv \vdash block_1\ block_2 \Rightarrow tenv_2}$$

$$[FUN_\pi]\quad \frac{\begin{array}{l} denv \vdash t : s \\ denv \vdash (p_{i1}, \ldots, p_{im}) : (t_1, \ldots, t_m) \Rightarrow tenv_i \quad \text{(FORALL } i) \\ denv,\ tenv[v \mapsto \overline{t}]tenv_i, \{v\} \vdash e_i : !^a\ t\ \&\ W_i \quad \text{(FORALL } i) \end{array}}{\begin{array}{l} denv,\ tenv \vdash (\texttt{FUN}\ v\ p_{11} \ldots p_{1m} \Rightarrow e_1\ \&\ \ldots \&\ v\ p_{n1} \ldots p_{nm} \Rightarrow e_n) \Rightarrow \\ \qquad tenv[v \mapsto \overline{t}] \end{array}}$$

WHERE $\overline{t} = t_1 \underset{\mathtt{T}}{\rightarrow} \ldots \underset{\mathtt{T}}{\rightarrow} t_m \underset{a}{\rightarrow} t$

IF $exh^{denv}_{t_1 \ldots t_m}([[p_{11}, \ldots, p_{1m}], \ldots, [p_{n1}, \ldots, p_{nm}]])$

IF $\forall (v\ e'_1 \ldots e'_k) \in W_i : (e'_1, \ldots, e'_k) <^\pi_a (p_{i1}, \ldots, p_{im})$ (FORALL i)

$$[GEN]\quad \frac{denv,\ tenv \vdash FUN \ldots \Rightarrow tenv'[v \mapsto ts]}{denv,\ tenv \vdash FUN \ldots \Rightarrow tenv'[v \mapsto \forall tav.ts]}$$

IF $tav \notin$ FTV $(tenv') \cup$ FAV $(tenv')$

Figure 8: Analysis of Blocks

because `n` $<_{\mathtt{T}}$ `Succ n`. If Ackerman's function had taken its parameters in the opposite order we would use the permutation $(2, 1)$. □

The final side condition in rule $[FUN_\pi]$ is the demand that all the patterns should be exhaustive. This is important when we try to determine that a function is total and will rule out the possibility that the semantics will fail (because neither $[\lambda_1]$ nor $[\lambda_2]$ is applicable); because of its impact on reliable programming this condition is imposed also in Standard ML [15]. We achieve this by means of the function exh that takes a list of patterns as argument, a list of types as subscript and the definition environment as superscript[2] An important invariant is that the i'th pattern in each list of patterns should have the i'th type. (In the notation of Figure 8 each p_{ij} should have type t_i.) One way to define exh is as a functional program in a Miranda-like notion as is done in Figure 10. The first clause allows a nice termination of the recursive calls whereas the third clause terminates any unwanted call. In the second clause the easy case is when all the patterns in head position are variables: then we just perform a recursive call on the remaining parts. As a notational convention we write p for a pattern, pp for a list of patterns and ppp for a list of lists of patterns. Also $p : pp$ denotes prepending the element p to the list pp and $pp_1 ++ pp_2$ denotes appending two lists. Turning to the harder case in the second clause we have to split the patterns according to the constructors that may produce elements of the type in question. For each such constructor c the list

[2] Another usage of termination analysis, not mentioned in the Introduction, is to ensure that functions defined in a programming language may indeed be used in specifications (using the traditional two - valued logic).

$$get_k\ c\ [p_1 : pp_1, \ldots, p_n : pp_n]$$

is a list of lists of "exploded patterns": if p_i consists of a constructor that is not c we disregard $p_i : pp_i$; if p_i is the constructor c we "explode" the pattern; and a variable is treated as the pattern $c\ x_1 \ldots x_k$ where $x_1, x_2, \ldots$ is some list of variables. Each of these lists must be checked for exhaustiveness. All these test must yield true and we express this using a Miranda-like ZF-expression and the function *and* that calculates the conjunction of a list of truth values.

$$denv \vdash v : t \Rightarrow [v \mapsto t]$$
$$\text{IF} \quad denv \vdash t : s$$

$$\frac{denv \vdash p_i : t_i \Rightarrow tenv_i \qquad (\text{FORALL } i = 1, \ldots, (m \geq 0))}{denv \vdash (p_1, \ldots, p_m) : (t_1, \ldots, t_m) \Rightarrow [\,]\ tenv_1 \ldots tenv_m}$$

$$\frac{denv \vdash p_i : t_i \Rightarrow tenv_i \qquad (\text{FORALL } i = 1, \ldots, (m \geq 0))}{denv \vdash c\ p_1 \ldots p_m : t \Rightarrow [\,]\ tenv_1 \ldots tenv_m}$$
$$\text{IF } denv \vdash denv(c) \geq t_1 \underset{\mathtt{T}}{\rightarrow} \ldots \underset{\mathtt{T}}{\rightarrow} t_m \underset{\mathtt{T}}{\rightarrow} t$$

Figure 9: Pattern Decomposition and Well-formedness

Example 5 In Ackerman's function we must test for exhaustiveness of the patterns. Write S for `Succ`, Z for `Zero` and N for `'Num`. The initial call is

$$exh_{NN}\ ([[Z,\mathtt{m}], [S\mathtt{n}, Z], [S\mathtt{n}, S\mathtt{m}]])$$

where we omit the definition environment that is as given in a previous example. The second case of the second clause applies and gives

$$and\ [exh_N[[\ \mathtt{m}\ ]], exh_{NN}[[\ \mathtt{n}, Z], [\ \mathtt{n}, S\mathtt{m}\ ]]]$$

where we treated the constructor Z first and S second. Now the first case of the second clause applies twice and yields

$$and\ [exh_{[\,]}[[\ ]], exh_N[[Z], [S\mathtt{m}\ ]]]$$

The first clause gives

$$and\ [true, exh_N[[Z], [S\mathtt{m}\ ]]]$$

and the second part of the second clause gives

$$and\ [true, and[exh_{[\,]}[[\ ]], exh_N[[\ \mathtt{m}\ ]]\ ]]$$

$$exh^{denv}_{[\,]}\ ([[\,],\ldots,[\,]]) = true$$

$$exh^{denv}_{t_1\ldots t_m}([p_1 : pp_1,\ldots,p_n : pp_n]) =$$

if all of $p_1,\ldots,p_n$ is a variable then

$exh^{denv}_{t_2\ldots t_m}\ ([pp_1,\ldots,pp_n])$

if some of p_i is a constructor applied to something then

$and\ ([exh^{denv}_{t'_1\ldots t'_k\ t_2\ldots t_m}\ (get_k\ c\ [p_1 : pp_1,\ldots,p_n : pp_n])$
$\mid\ c\ \leftarrow$ constructors in $denv$ of type $\geq t'_1 \underset{\mathtt{T}}{\rightarrow} \cdots \underset{\mathtt{T}}{\rightarrow}\ t'_k \underset{\mathtt{T}}{\rightarrow} t_1])$

$exh^{denv}_{t_1\ldots t_m}\ ([pp_1,\ldots,pp_n]) = false$ if no previous case applies

$$get_k\ c\ [\,] = [\,]$$

$$get_k\ c\ ((p : pp) : ppp) =$$

if p is $c\ p_1\ldots p_k$ then

$([p_1,\ldots,p_k]\ {+}{+}pp) : (get_k\ c\ ppp)$

if p is $c'\ p_1\ldots p_l$ for $c \neq c'$ then

$get_k\ c\ ppp$

if p is a variable then

$([x_1,\ldots,x_k]\ {+}{+}pp) : (get_k\ c\ ppp)$

Figure 10: Checking Exhaustiveness

This all evaluates to *true* showing that the patterns are exhaustive. (They are also mutually exclusive but this is not specified by *exh*.) □

Putting all this together we have now obtained the annotation and type for Ackerman's function that was claimed in Section 2:

$$!^{\mathtt{T}}\ \mathtt{'Num} \underset{\mathtt{T}}{\rightarrow} \mathtt{'Num} \underset{\mathtt{T}}{\rightarrow} \mathtt{'Num}$$

Also it should be clear that our technique applies to all functions that strictly adhere to *primitive recursive form*. For a third and much simpler example consider

```
FUN twice f x = f (f x)
```

that may be analysed so as to obtain the type environment

$$[\ \mathtt{twice} \mapsto \forall\ \mathtt{'a}\,.\forall\ \mathtt{'b}\,.\forall\ \mathtt{'1}\,.(\mathtt{'a} \underset{\mathtt{'1}}{\rightarrow} \mathtt{'b}) \underset{\mathtt{T}}{\rightarrow} \mathtt{'a} \underset{\mathtt{'1}}{\rightarrow} \mathtt{'b}]$$

Thus `twice` may be instantiated to apply to a total function as well as a possibly partial function and in both cases we will have as much information about the result as about the argument.

Remark A final important point is that we have only specified the analysis for those language constructs that are accessible to the programmer. This means that temporary expressions (i.e. λ-expressions) and semantic extensions (e.g. $\mathtt{FUN}_k$) are of no concern to the analysis. However, we shall see in the next section that they will be of concern for the definition of validity. □

4 Soundness

To prove the soundness of the analysis we must decide on a notion of validity for the judgements that the analysis deals with. For this we begin with values and expressions of closed types (i.e. monotypes), extend to closed type schemes and then finally allow free type and annotation variables.

Validity

For values of a simple (i.e. containing no function space) algebraic and closed type $t_1 \dots t_p\ tc$ we set:

$$\begin{array}{rl} \models_{denv} w : t_1 \dots t_p\ tc \equiv & \exists c, w_1, \dots, w_m, t'_1, \dots, t'_m : \\ & w = c\ w_1 \dots w_m \wedge \\ & denv \vdash denv(c) \geq t'_1 \underset{\mathtt{T}}{\rightarrow} \dots \underset{\mathtt{T}}{\rightarrow} t'_m \underset{\mathtt{T}}{\rightarrow} t_1 \dots t_p\ tc \wedge \\ & \forall i \leq m : \models_{denv}\ w_i : t'_i \end{array}$$

Since $t_1 \dots t_p\ tc$ is *closed* also $t'_1, \dots, t'_m$ will be closed due to the premises of rule $[DEF]$ in Figure 5; and since $t_1 \dots t_p\ tc$ is *simple* the definition is well-defined by induction in the structure of the value w (assuming $FV(w) = \emptyset$). Also $t'_1, \dots, t'_m$ are determined uniquely due to the premises of rule $[DEF]$ in Figure 5.

For expressions of a simple and closed type we use the notion of logical relations:

$$\models^{\rho}_{denv}\ e : !^{\epsilon}\ t \equiv (\exists w : \rho \vdash e \Downarrow w) \Rightarrow \models^{\rho}_{denv} e : !^{\mathtt{T}}\ t$$

$$\begin{array}{l} \models^{\rho}_{denv}\ e : !^{\mathtt{T}}\ t_1 \underset{a}{\rightarrow} t_2 \equiv \exists w : (\rho \vdash e \Downarrow w \wedge \\ \qquad\qquad \forall \rho' : \forall e' : \ (\rho', e')\ \text{pure} \wedge \models^{\rho'}_{denv}\ e' : !^{\mathtt{T}}\ t_1 \\ \qquad\qquad\qquad \Rightarrow \models^{\rho'}_{denv}\ w\ e' : !^{a}\ t_2) \end{array}$$

$$\models^{\rho}_{denv} e : !^{\mathtt{T}}\ t_1 \dots t_p\ tc \equiv \exists w : (\rho \vdash e \Downarrow w \wedge \models_{denv} w : t_1 \dots t_p\ tc)$$

This definition is well-defined by induction in the structure of the type; more precisely we prove well-definedness of $\models^{\rho}_{denv}\ e : !^{a}\ t$ by induction in (t, a) ordered lexicographically: the first component by substructure and the second by $\mathtt{T}$ below ϵ. Note the choice of a new fresh environment together with the argument expression in the clause for function space. Also note that if we did not insist on types being simple this straightforward inductive argument for well-definedness would no longer work: in the definition of validity of values

we should then expect to meet a value of a function type and for this the notion of logical relations would have to be used once again. We return to this issue in the Conclusion.

It is helpful with a few simple facts about the notion of validity.

Fact 6 For a value w as produced by $\rho' \vdash e' \Downarrow w$ we have
$\models^{\rho}_{denv}\ \ w : !^a\ t \Leftrightarrow \models^{[\,]}_{denv}\ \ w : !^a\ t$.

Proof Inspection of the three clauses in the definition of validity and using that a value evaluates to itself in all environments. □

Fact 7 If $\rho \vdash e \Downarrow w$ then $\models^{\rho}_{denv}\ \ e : !^a\ t \Leftrightarrow \models^{\rho}_{denv}\ \ w : !^a\ t$.

Proof Induction on (t, a) ordered lexicographically as above and using that the semantics is deterministic and that a value evaluates to itself. □

Corollary 8 (*Semantic Equivalence*)
If $\rho_1 \vdash e_1 \Downarrow w \wedge \rho_2 \vdash e_2 \Downarrow w$ or if $\rho_1 \vdash e_1 \Downarrow w \Leftrightarrow\ \ \rho_2 \vdash e_2 \Downarrow w$ then
$\models^{\rho_1}_{denv}\ \ e_1 : !^a\ t \Leftrightarrow \models^{\rho_2}_{denv}\ \ e_2 : !^a\ t$.

Proof If e_1 and e_2 both evaluate use the above two facts. If neither e_1 nor e_2 evaluates then the result is immediate. □

To deal with environments we first extend the definition of validity to include closed type schemes:

$$\begin{array}{l}\models^{\rho}_{denv}\ \ e : !^a\ \forall tav_1 \ldots \forall tav_n.\ t \equiv \\ \quad \forall U :\ U \text{ is a ground and well-formed substitution with domain } \{tav_1, \ldots, tav_n\} \\ \qquad \Downarrow \\ \qquad\quad \models^{\rho}_{denv}\ \ e : !^a\ U(t)\end{array}$$

For environments we then have

$$\begin{array}{ll}\models_{denv} \rho : tenv \equiv & \text{dom}\ (\rho) = \text{dom}\ (tenv) \wedge \\ & \forall v \in \text{dom}\ (\rho) : \models^{\rho}_{denv}\ \ v : !^{\mathrm{T}}\ tenv(v)\end{array}$$

where we restrict $tenv$ to map variables to closed type schemes only.

Expressions

Turning to the judgements for expressions we now allow free type and annotation variables:

$$\begin{array}{l}denv,\ tenv,\ cenv \models e : !^a\ t\ \&\ W \equiv \\ \quad \forall U : \forall \rho : (U \text{ is a ground and well-formed substitution that covers } tenv,\ a,\ t \\ \qquad\qquad \wedge \models_{denv} \rho : U(tenv)) \\ \qquad \Downarrow \\ \qquad\quad \models^{\rho}_{denv} e : !^{U(a)}\ U(t) \wedge \\ \qquad\quad \forall v, w, e_1, \ldots, e_n : \text{ if } v\ e_1 \ldots e_n \text{ is a maximal and exposed} \\ \qquad\qquad\qquad\qquad\quad \text{call in } \rho \vdash e \Downarrow w \text{ and } v \in cenv \text{ then} \\ \qquad\qquad\qquad\qquad\quad (v\ e_1 \ldots e_n) \in W\end{array}$$

This requires a few auxiliary notions. A call $v\ e_1 \ldots e_n$ *occurs* in $\rho \vdash e \Downarrow w$ if there is a subinference of the form $\rho' \vdash v\ e_1 \ldots e_n \Downarrow w'$. The call is *maximal* if it is not part of a strictly larger call $v\ e_1 \ldots e_n\ e_{n+1} \ldots e_q$ that occurs in $\rho \vdash e \Downarrow w$. The call is *exposed* if there are no occurrences of rules $[\lambda_1]$ or $[\lambda_2]$ of Figure 4 on the way from the root of $\rho \vdash e \Downarrow w$ to the root of $\rho' \vdash v\ e_1 \ldots e_n \Downarrow w'$. This latter condition restricts us from looking inside the bodies of nested calls; also the exposed part of an inference has the same environment everywhere.

Lemma 9 *(Soundness of expressions)* $denv,\ tenv,\ cenv \vdash e : !^a\ t\ \&\ W$ implies $denv,\ tenv,\ cenv \models e : !^a\ t\ \&\ W$.

Proof We proceed by induction on the inference tree and let a ground substitution U that is $denv$-well-formed and that covers $tenv$, a and t be given as well as an environment ρ that satisfies $\models \rho : U(tenv)$. The proof then amounts to inspecting each of the rules and axioms of Figure 7. Please refer to the Appendix for the details of this and subsequent proofs. □

Matching

Soundness of blocks requires several preparations. For matching we have two results. One shows that when matching succeeds it produces a correct environment. The other shows that exhaustiveness of the patterns prevents matching from failing.

Lemma 10 (*Soundness of matching*) Let U be a ground and $denv$-well-defined substitution that covers $tenv, t_1, \ldots, t_m$. If

$$(p_1, \ldots, p_m) : (w_1, \ldots, w_m) \leadsto \rho$$
$$denv \vdash (p_1, \ldots, p_m) : (t_1, \ldots, t_m) \Rightarrow tenv$$
$$\forall i : \models^{[\,]}_{denv}\ w_i : !^{\mathsf{T}}\ U(t_i)$$

then $\models_{denv} \rho : U(tenv)$.

Proof We proceed by induction on the syntax of the tuple of patterns $(p_1, \ldots, p_m)$. One case is when $(p_1, \ldots, p_m)$ is a variable, another is when it is a constructor applied to a number of patterns, and the third is when it is a proper tuple of patterns. □

Lemma 11 (*Soundness of exhaustiveness*) Let U be a ground and $tenv$-well-formed substitution that covers $t_1, \ldots, t_m$. If

$$exh^{denv}_{t_1,\ldots,t_m}\ ([[p_{11}, \ldots, p_{1m}], \ldots, [p_{n1}, \ldots, p_{nm}]])$$

$$\forall i : \models^{[\,]}_{denv}\ w_i : !^{T}\ U(t_i)$$

then matching must succeed, i.e.

$$\neg \forall i : (p_{i1}, \ldots, p_{im}) : (w_1, \ldots, w_m) \not\leadsto$$

Proof We define a well-founded order $\prec$ by $(w_1, \ldots, w_m) \prec (w'_1, \ldots, w'_n)$ iff the sum of the sizes of $w_1, \ldots, w_m$ is strictly less than the sum of the sizes of $w'_1, \ldots, w'_n$. We then prove the result by contradiction in an induction on $(w_1, \ldots, w_m)$ ordered by $\prec$. □

Syntactic continuity and inductiveness

For the partial correctness part of functions in the soundness of blocks it is convenient to be able to perform a numerical induction on the number of times a function is unfolded. This is facilitated by the function value

$$\texttt{FUN}_k\ \rho\ v\ \vec{p_1} \Rightarrow \overline{e}_1\ \&\ \ldots \&\ v\ \vec{p_n} \Rightarrow \overline{e}_n$$

that allows only k unfoldings. Its formal semantics was dealt with in Section 2.

We may now define a syntactic ordering $\sqsubseteq$ upon terms and judgements extended with $\texttt{FUN}_k$. The basic axioms may informally be stated as

$$\texttt{FUN}_k \sqsubseteq \texttt{FUN}_l \quad \text{if } k \leq l$$

$$\texttt{FUN}_k \sqsubseteq \texttt{FUN}$$

and where $\sqsubseteq$ is then extended in a componentwise manner to a partial order on expressions, values and environments. We trust the details are obvious and else refer to [18, Chapter 6].

To motivate our auxiliary results it is helpful to pretend that we are doing denotational semantics. For this define the semantic function *sem* as follows:

$$sem(\rho, e) = \begin{cases} w & \text{if } \rho \vdash e \Downarrow w \\ \bot & \text{if } \neg\exists w : \rho \vdash e \Downarrow w \end{cases}$$

Here $\bot$ is a new symbol and it may be incorporated into the partial order by setting $\bot \sqsubseteq e$ for all expressions e. The function *sem* is well-defined because the semantics of Section 2 is deterministic. In denotational semantics a major result would then be to establish the continuity of *sem*.

The monotonicity part amounts to:

Lemma 12 (*Monotonicity*) If $\rho_1 \sqsubseteq \rho_2, e_1 \sqsubseteq e_2$ and $\rho_1 \vdash e_1 \Downarrow w_1$ then there exists w_2 such that $\rho_2 \vdash e_2 \Downarrow w_2$ and $w_1 \sqsubseteq w_2$; provided that all (ρ_i, e_i) are pure.

Proof We proceed by induction on the inference tree for $\rho_1 \vdash e_1 \Downarrow w_1$. □

The other half of the continuity result is a bit harder. First, we define the notion of *labelling*. Given an expression e we may obtain a labelling[3] e' of e by replacing all `FUN` in e by $\texttt{FUN}_k$'s (where the subscript need not be the same for different occurrences). The labelling is *safe for m* if all subscripts are chosen to be greater than or equal to m. If all subscripts are chosen to be exactly m we write e' as $e^{[m]}$. Similar notions apply to

[3]Note that a labelling corresponds to a compact (or finite) element in domain theory.

environments and values. Finally, the *size* of $\rho \vdash e \Downarrow w$ is the number of nodes in the inference tree.

Lemma 13 (*Continuity*) Let m be arbitrary and let (ρ, e) and (ρ', e') be pure. If $\rho \vdash e \Downarrow w$ and (ρ', e') is a labelling of (ρ, e) that is safe for m plus the size of $\rho \vdash e \Downarrow w$, there exists a labelling w' of w such that $\rho' \vdash e' \Downarrow w'$ and w' is safe for m.

Proof We proceed by induction on the inference tree for $\rho \vdash e \Downarrow w$. □

For the continuity of *sem* let $(\rho_k, e_k)_k$ be a pure chain with pure limit (ρ, e). If $sem(\rho, e) = \bot$ we are done by monotonicity so assume $\rho \vdash e \Downarrow w$ and that this inference tree has size m_0. The chain $(\rho^{[m+m_0]}, e^{[m+m_0]})_m$ clearly has limit (ρ, e) and by the above lemmas there exists w_m such that $\rho^{[m+m_0]} \vdash e^{[m+m_0]} \Downarrow w_m$ and $w^{[m]} \sqsubseteq w_m \sqsubseteq w$. Furthermore, $(w^{[m]})_m$ is clearly a chain with limit w. Each element in the chain $(\rho^{[m+m_0]}, e^{[m+m_0]})_m$ is dominated by some element in the chain (ρ_k, e_k). By monotonicity each w_m is dominated by some $sem(\rho_k, e_k)$ and clearly $sem(\rho_k, e_k) \sqsubseteq w$. This shows that $(sem(\rho_k, e_k))_k$ is a chain with limit w. This may be summarized as:

Corollary 14 *sem* is continuous (on pure arguments). □

A more formal, and considerably more tedious, development along these lines may be found in [18, pages 177-184].

Continuing our excursion into denotational semantics the notion of inclusive predicate is useful.

Lemma 15 (*Inclusiveness*) Let a and t be closed annotations and types, respectively, with t being *denv*-well-formed. If $(\rho_k, e_k)_k$ is a pure chain with pure limit (ρ, e) and if

$$\forall k : \models^{\rho_k}_{denv} \; e_k : !^a \; t$$

then $\models^{\rho}_{denv} \; e : !^a \; t$.

Proof We proceed by induction on (t, a) ordered lexicographically and using the continuity of *sem*. □

The final result is of a somewhat different character and is related to the total correctness part of functions.

Fact 16: For all definition environments *denv* obtainable from Figure 5 and for all well-formed and closed types t, i.e. $denv \vdash t : s$, there exists a pure value w_t such that $\models^{[\,]}_{denv} \; w_t : !^{\mathrm{T}} \; t$.

Proof We proceed by induction on t. For function types we construct a constant function. For algebraic types we use induction on the order in which they were introduced into *denv*; we here use the premiss in rule [*DEF*] of Figure 5 that guarantees the non-emptiness of types. □

Blocks and programs

Turning to the judgements for blocks we again allow free type and annotation variables:

$$denv,\; tenv \models block \Rightarrow tenv' \equiv$$

$$\forall U : \forall \rho : (U \text{ is a ground and } denv\text{-well-formed substitution that covers } tenv' \wedge \models_{denv} \rho : U(tenv) \wedge \rho \text{ is pure})$$
$$\Downarrow$$
$$\exists \rho' : \rho \vdash block \Downarrow \rho' \wedge \models_{denv} \rho' : U(tenv') \wedge \rho' \text{ is pure}$$

For this to be meaningful it must be the case that U also covers $tenv$; that this is the case follows from:

Fact 17 If $denv,\ tenv \vdash block \Rightarrow tenv'$ then $tenv'$ is a possibly trivial prolongation of $tenv$.

Proof We proceed by induction on the inference. □

Lemma 18 *(Soundness of blocks)* $denv,\ tenv \vdash block \Rightarrow tenv'$ implies $denv,\ tenv \models block \Rightarrow tenv'$.

Proof We proceed by induction on the inference tree. (We do not let U and ρ be given a priori for all cases because U must be varied in case $[GEN]$.) The harder case is the one for function definition where we have two cases. When the function must be shown to be possibly partial we use the constrained functions $\texttt{FUN}_k$ and induction in k; we then pass to the limit using the lemma on inclusiveness. When the function must be shown to be total we used the forced functions $\texttt{FUN}\,[\ldots]$ and prove termination using the well-founded order $<^{\pi}_{\mathtt{T}}$ of $[\texttt{FUN}_\pi]$. □

The overall correctness of the analysis is then given by:

Theorem 19 Let $\vdash denv\ block\ e : !^a\ t$, write $denv$ for the definition environment obtained from $defn$, and let U be a ground and $denv$-well-formed substitution that covers a and t. We then have:

- if $Ua = \mathtt{T}$ there exists w such that $\vdash denv\ block\ e \Downarrow w$
- if $\vdash defn\ block\ e \Downarrow w$ then $\models^{[\,]}_{denv}\ w : !^{\mathtt{T}}\ U(t)$ □

5 Conclusion

We have developed an approach for the termination analysis of higher-order functional programs and proved it sound. We believe that a pleasant aspect of our approach is to make the result of the analysis available to programmers in a readable form: types are well-established and we have shown that a rather minimal extension of the syntax of types allows for conveying the termination information.

For annotated types we decided to label the function space constructor rather than individual types. For strictness analysis both approaches have been used: an example of the former is [29] and an example of the latter is [11]. In terms of readability in applications we believe our choice is superior and is sufficiently readable to be incorporated into the syntax of a realistic programming language. However, if we shift from an eager language to a lazy language the other possibility will be more informative; to see this note that a thunk like $\texttt{'Unit} \rightarrow t$ is written simply t in a lazy language and to distinguish between $\texttt{'Unit} \underset{\mathtt{T}}{\rightarrow} t$ and $\texttt{'Unit} \underset{\epsilon}{\rightarrow} t$ we must allow writing $!^{\mathtt{T}}\ t$ and $!^{\epsilon}\ t$, respectively.

We conjecture that it would be possible to extend this work with *simple* run-time complexity information for functions traversing data structures in a regular and systematic way. Examples include the `map`, `fold`, and `filter` functions from functional languages. The idea is that linear run-time is ensured if the function call in question always recurses by decreasing one of the parameters. We believe that such functions occur frequently enough in functional programming that the results obtainable may be useful for improving the implementation and for annotating high-level functions built in terms of such primitives. While we have mainly worst case complexity in mind it would be interesting to investigate whether the techniques of [9] for average-case analysis may be incorporated into our type based approach.

Extensions and limitations Extensions are needed to handle functions counting up to a treshold rather than down to zero. Similarly a more complicated well-founded order [6] (involving ranking functions on the constructors of algebraic data types) will be needed in order to handle the `flatten` function that takes a list of trees and produces a list of leaves by recursively decomposing a composite tree at the head of the list to its list of (simpler!) subtrees. Finally our analysis is very sensitive to the textual appearance of subpatterns: in the definition of Ackerman's function it is crucial that the first recursive call is written `ack n (Succ Zero)` rather than the "equivalent" `ack (id n) (Succ Zero)` where `id` is the identity defined by `id x => x`. To handle the latter we would need to add a new component to the analysis: perhaps a "second-order" component using "quantity names" to be able to express the relationship between the result and the argument of a function.

We need to investigate the possibility of a constraint-based algorithm for automatically inferring the annotated type information. This is likely to give faster results than adapting the algorithm $\mathcal{W}$ as is briefly discussed in the Appendix. However, we do believe that the analysis is manageable both from a theoretical point of view (by being applicable to all functions that strictly adhere to primitive recursive form) and practically (by being not too much more costly than algorithm $\mathcal{W}$).

Proof techniques for operational semantics This research is part of an undertaking towards studying the applicability of operational semantics and inference systems as the basis for reasoning about program analyses and program transformations and the development has been structured so as to highlight the principles we see emerging.

To cater for the soundness proof we were twice compelled to a piecemeal extension of the syntax and semantics. The first instance was the introduction of $\mathtt{FUN}_k$ so as to facilitate proving partial correctness statements by induction on the number of recursive unfoldings. The second instance was the introduction of $\mathtt{FUN}[\prec, \vec{w}, \overline{w}]$ to facilitate proving total correctness statements by induction on the arguments. New extensions might be conceivable for other methods of proofs required for other results; hence a piecemeal extension of the syntax and semantics (but not the inference system!) seems unavoidable. It is important to stress that when doing so the relevant proofs have to be amended due to the new rules present in the semantics. It remains open whether these problems might be alleviated by adopting other forms of structural operational semantics: one possibility is small-step operational semantics [20] and another is "GooSOS" that allows explicit specification of

infinite as well as finite behaviours [5].

Since this is a phenomenon *not* found in denotational semantics it may be appropriate to ask why we insisted on an operational semantics. Similarly one may ask why we favoured the inference system approach over abstract interpretation. In both cases the answer is that there are language constructs like concurrency for which denotational semantics is no easy task (to put it mildly). While this is not a concern of the present paper it would be a concern if the termination analysis was to be used for some of the applications mentioned in the Introduction and our future work is likely to follow this path. Furthermore, proponents of inference systems for strictness analysis often claim that they give a cleaner separation between specification and implementation than does abstract interpretation. It is hard to be objective about such claims but we believe that our inference system for termination analysis compares favourably with the abstract interpretation for quasi-termination developed in [10].

Acknowledgement This work was supported in part by the European Union (ESPRIT BRA project 8130 LOMAPS, and ESPRIT BRA working group 6809 SEMANTIQUE) and by the Danish Science Research Council (project DART).

References

[1] M. Baudinet: Proving Termination Properties of PROLOG Programs: A Semantic Approach, *Proc. Logic in Computer Science*, pp. 336–347, (1988).

[2] R.Berghammer, B.Elbl, U.Schmerl: Proving Total Correctness of Programs in Weak Second-Order Logic, *Proc. Semantics: Foundations and Applications* (1992), SLNCS **666** pp. 51–72, (1993).

[3] B.Bjerner, S.Holmström: A compositional approach to time analysis of first order lazy functional languages, *Proc. ACM Conf. Functional Programming and Computer Architecture*, pp. 157–165 (with errata sheet), ACM Press, (1989).

[4] P.Cousot, N.Halbwachs: Automatic Discovery of Linear Restraints among Variables of a Program, *Proc. ACM Conf. on Principles of Programming Languages*, pp. 84–96, ACM Press, (1978).

[5] P.Cousot, R.Cousot: Inductive Definitions, Semantics and Abstract Interpretation, *Proc. ACM Conf. on Principles of Programming Languages*, ACM Press, (1992).

[6] N.Dershowitz: Orderings for Term-Rewriting Systems. *Theoretical Computer Science* **17**, pp. 279–310, (1982).

[7] V.Dornic, P.Jouvelot, D.K.Gifford: Polymorphic Time Systems for Estimating Program Complexity, *ACM Letters on Programming Languages and Systems*, **1** *1*, pp. 33–45, (1992).

[8] S.Feferman: Logics for termination and correctness of functional programs, *Proc. Leeds Proof Theory*, (1990).

[9] P.Flajolet, B.Salvy, P.Zimmermann: Automatic average-case analysis of algorithms, *Theoretical Computer Science* **79**, pp. 37–109, (1991).

[10] C.K.Holst: Finiteness Analysis, *Proc. FPCA'91*, SLNCS **523**, pp. 473–495, (1991).

[11] T.M.Kuo, P.Mishra: Strictness Analysis: A New Perspective based on Type Inference. *Proc. FPCA'89*, pp. 260–272, ACM Press, (1989).

[12] D.LeMetayer: ACE: An Automatic complexity evaluator, *ACM Transactions of Programming Languages*, **10** *2*, pp. 248–266, (1988).

[13] U. Martin, E.Scott: The order types of termination orderings on monadic terms, strings and multisets, *Proc. Logic in Computer Science*, pp. 356–363, (1993).

[14] R. Milner: A Theory of Type Polymorphism in Programming, *Journal of Computer Systems* **17**, pp. 348–375, (1978).

[15] R.Milner, M.Tofte, R.Harper: *The Definition of Standard ML*, MIT Press, (1990).

[16] H.R.Nielson: A Hoare-like Proof System for Total Correctness of Nested Recursive Procedures, *Proc. Fourth Hungarian Computer Science Conference*, (1985).

[17] H.R.Nielson: Proof Systems for Computation Time, *Proc. Third Conf. on Foundations of Software Technology and Theoretical Computer Science*, (1983).

[18] F.Nielson, H.R.Nielson: *Two-Level Functional Languages*, Cambridge Tracts in Theoretical Computer Science **34**, (1992).

[19] F. Nielson, H.R. Nielson: From CML to Process Algebras, *Proc. CONCUR'93*, SLNCS **715**, pp. 493–508, 1993.

[20] G.D.Plotkin: Structural Operational Semantics, *Lecture Notes DAIMI FN-19*, Aarhus University, Denmark, (1981, reprinted 1991).

[21] M.Rosendahl: Automatic Complexity Analysis, *Proc. ACM Conf. on Functional Programming and Computer Architecture*, pp. 144–156, ACM Press, (1989).

[22] D. Sands: Calculi for Time Analysis of Functional Programs, *Ph.D.-dissertation*, Imperial College, University of London, (1990).

[23] D.Sands: Time Analysis, Cost Equivalence and Program Refinement, *Proc. Foundations of Software Technology and Theoretical Computer Science*, SLNCS **560**, pp. 25-39, (1991).

[24] D.Sands: Complexity Analysis for a Lazy Higher-Order Language, *Proc. ESOP'90*, SLNCS **432**, pp. 361–376, (1990).

[25] J.-P. Talpin, P. Jouvelot: The Type and Effect Discipline, *Proc. LICS'92*, 1992. (Also see *Information and Computation* **111** *2*, 1994.)

[26] Yan-Mei Tang: Control-Flow Analysis by Effect Systems and Abstract Interpretation, *Ph.D.-thesis*, Ecole Nationale Superieure des Mines de Paris, (1994).

[27] D.A.Turner: Miranda: A Non-strict Functional Language with Polymorphic Types, *Proc. FPCA'85*, SLNCS **201**, pp. 1–16, (1985).

[28] C.Walther: Argument-bounded Algorithms as a Basis for Automated Termination Proofs, *Proc. CADE-9*, SLNCS **310**, pp. 602–621, (1988).

[29] D.A.Wright: A New Technique for Strictness Analysis, *Proc. TAPSOFT'91*, SLNCS **494**, pp. 235–258, (1991).

Appendix

Proof of Lemma 9 *(Soundness of expressions)*

We proceed by induction on the inference tree and let a ground substitution U that is $denv$-well-formed and that covers $tenv, a$ and t be given as well as an environment ρ that satisfies $\models \rho : U(tenv)$. The proof then amounts to inspecting each of the rules and axioms of Figure 7.

The case [*VAR*]. For the first conjunct we know $U(a) = \mathtt{T}$ and from the assumptions we have $\models^{\rho}_{denv}\ v : !^{\mathsf{T}}\ U(tenv)(v)$. Since $denv \vdash tenv(v) \geq t$ we also have $denv \vdash U(tenv)(v) \geq U(t)$ and hence the desired $\models^{\rho}_{denv}\ v : !^{\mathsf{T}}\ U(t)$ follows.

For the second conjunct we know that if $v \in cenv$ then v is the only maximal call in $\rho \vdash v \Downarrow w$ and clearly it is exposed.

The case [*CON*]. For the first conjunct we know $U(a) = \mathtt{T}$. It suffices to establish $\models^{\rho}_{denv}\ c : !^{\mathsf{T}}\ U(denv(c))$ since the desired $\models^{\rho}_{denv}\ c : !^{\mathsf{T}}\ U(t)$ then follows as in [*VAR*]. Since $denv(c)$ is closed we can dispense with U and simply use:

Fact 20 $\models^{\rho}_{denv}\ c : !^{\mathsf{T}}\ denv(c)$

Proof Consider $denv \vdash denv(c) \geq t'_1 \underset{\mathsf{T}}{\rightarrow} \cdots \underset{\mathsf{T}}{\rightarrow} t'_m \underset{\mathsf{T}}{\rightarrow} t_1 \ldots t_p\ tc$ with the latter type being closed and show

$$\models^{\rho}_{denv}\ c : !^{\mathsf{T}}\ t'_1 \underset{\mathsf{T}}{\rightarrow} \cdots \underset{\mathsf{T}}{\rightarrow} t'_m \underset{\mathsf{T}}{\rightarrow} t_1 \ldots t_p\ tc$$

This boils down to assuming $\models^{\rho_i}_{denv}\ e_i : !^{\mathsf{T}}\ t'_i$ with $\rho_i \vdash e_i \Downarrow w_i$ and proving

$$\models^{\rho}_{denv}\ c\ w_1 \ldots w_m : !^{\mathsf{T}}\ t_1 \ldots t_p\ tc$$

But this is immediate since $\models_{denv} w_i : t'_i$. □

The second conjunct is immediate since $c \notin cenv$.

The case [*APP*]. For the first conjunct we know from the induction hypothesis that $\models^{\rho}_{denv}\ e_1 : !^{U(a)}\ U(t_1 \underset{a}{\rightarrow} t_2)$ and $\models^{\rho}_{denv}\ e_2 : !^{U(a)}\ U(t_1)$ and we must show that $\models^{\rho}_{denv}\ e_1\ e_2 : !^{U(a)}\ U(t_2)$.

If $U(a) = \mathtt{T}$ we have $\rho \vdash e_1 \Downarrow w_1$ for some w_1 and furthermore $\models^{\rho}_{denv} w_1\ e_2 : !^{\mathsf{T}}\ U(t_2)$. From the Corollary 8 on Semantic Equivalence it is then immediate that also $\models^{\rho}_{denv} e_1\ e_2 : !^{\mathsf{T}}\ U(t_2)$.

If $U(a) = \epsilon$ we must assume that $\rho \vdash e_1\ e_2 \Downarrow w$ and then $\models^{\rho}_{denv} e_1\ e_2 : !^{\mathsf{T}}\ U(t_2)$ has to be shown. In this case also $\rho \vdash e_1 \Downarrow w_1$ and $\rho \vdash e_2 \Downarrow w_2$ for some w_1 and w_2 and then $\models^{\rho}_{denv}\ e_1 : !^{\mathsf{T}}\ U(t_1 \underset{\mathsf{T}}{\rightarrow} t_2)$ and $\models^{\rho}_{denv} e_2 : !^{\mathsf{T}}\ U(t_1)$. We may then proceed as in the case $U(a) = \mathtt{T}$.

For the second conjunct let $v\ e'_1 \ldots e'_n$ be a maximal and exposed call in $\rho \vdash e_1\ e_2 \Downarrow w$ with $v \in cenv$. The last rule used in $\rho \vdash e_1\ e_2 \Downarrow w$ must be either $[APP_C]$, $[APP^{<}_{FUN}]$ or $[APP^{=}_{FUN}]$.

If it is $[APP_C]$ or $[APP^{<}_{FUN}]$ we have $\rho \vdash e_1 \Downarrow w_1$ and $\rho \vdash e_2 \Downarrow w_2$ with $w = w_1\ w_2$. The call $v\ e'_1 \ldots e'_n$ may be all of $\rho \vdash e_1\ e_2 \Downarrow w$ (except for $[APP_C]$), may be a proper part of $\rho \vdash e_1 \Downarrow w_1$ or may be a part of $\rho \vdash e_2 \Downarrow w_2$. In the first case there will also be a maximal call $v\ e'_1 \ldots e'_{n-1}$ in $\rho \vdash e_1 \Downarrow w_1$ and hence $e_1 = v\ e'_1 \ldots e'_{n-1} \in W_1$ so $e_1\ e_2 = v\ e'_1 \ldots e'_n \in W$. In the second case $v\ e'_1 \ldots e'_n \in W_1 \backslash \{e_1\} \subseteq W$ and in the third case $v\ e'_1 \ldots e'_n \in W_2 \subseteq W$.

If the last rule used in $\rho \vdash e_1\ e_2 \Downarrow w$ is $[APP^{=}_{FUN}]$ we additionally have to consider the possibility that $v_1\ e'_1 \ldots e'_n$ is a maximal and exposed call in

$$\overline{\rho}[v \mapsto \ldots] \vdash (\lambda \ldots) \ldots \Downarrow w$$

However, our definition of "exposed" prevents this from being the case.

The case $[IF]$. For the first conjunct suppose first that $U(a) = \mathtt{T}$. From $\models^{\rho}_{denv} e_1 : !^{\mathsf{T}}\ \mathtt{'Bool}$ it is immediate that $\rho \vdash e_1 \Downarrow w_1$ where $w_1 \in \{\mathtt{True}, \mathtt{False}\}$. Let

$$i_1 = \begin{cases} 2 & w_1 = \mathtt{True} \\ 3 & w_1 = \mathtt{False} \end{cases}$$

so that

$$\rho \vdash \mathtt{IF}\ e_1\ \mathtt{THEN}\ e_2\ \mathtt{ELSE}\ e_3 \Downarrow w \quad \text{ iff } \rho \vdash e_{i_1} \Downarrow w$$

From $\models^{\rho}_{denv} e_{i_1} : !^{\mathsf{T}}\ U(t)$ and the Corollary 8 on Semantic Equivalence we then have the desired $\models^{\rho}_{denv} \mathtt{IF}\ e_1\ \mathtt{THEN}\ e_2\ \mathtt{ELSE}\ e_3 :!^{\mathsf{T}}\ U(t)$.

Next suppose that $U(a) = \epsilon$. We must assume that $\rho \vdash \mathtt{IF}\ e_1\ \mathtt{THEN}\ e_2\ \mathtt{ELSE}\ e_3 \Downarrow w$ and then show $\models^{\rho}_{denv} \mathtt{IF}\ e_1\ \mathtt{THEN}\ e_2\ \mathtt{ELSE}\ e_3 : !^{\mathsf{T}}\ U(t)$. In this case also $\rho \vdash e_1 \Downarrow w_1$ and with i_1 as above $\rho \vdash e_{i_1} \Downarrow w$. Hence $\models^{\rho}_{denv} e_1 : !^{\mathsf{T}}\ \mathtt{'Bool}$ and $\models^{\rho}_{denv} e_{i_1} : !^{\mathsf{T}}\ U(t)$ and we may proceed as in the case $U(a) = \mathtt{T}$.

For the second conjunct we simply use the induction hypothesis because any call $v\ e'_1 \ldots e'_n$ must be part of one of e_1, e_2 or e_3.

The case $[SUB]$. The first conjunct is a consequence of the following facts:

Fact 21 If $a \rightsquigarrow a'$ and $\models^{\rho}_{denv} e : !^{U(a)}\ U(t)$ then $\models^{\rho}_{denv} e : !^{U(a')}\ U(t)$

Proof We proceed by induction on $a \rightsquigarrow a'$ and the only non-trivial case is when $a = \mathtt{T}$ and a' is arbitrary. If $U(a') = \mathtt{T}$ the result is immediate (since $U(a) = U(a')$) and if $U(a') = \epsilon$ we use the definition of validity. □

Fact 22 If $t \rightsquigarrow t'$ and $\models^{\rho}_{denv} e : !^{U(a)}\ U(t)$ then $\models^{\rho}_{denv} e : !^{U(a)}\ U(t')$.

Proof We proceed by induction on $t \rightsquigarrow t'$ and the only non-trivial case is when $t = t_1 \underset{a_0}{\rightarrow} t_2$ and $t' = t'_1 \underset{a'_0}{\rightarrow} t'_2$ where $t_1 = t'_1, t_2 \rightsquigarrow t'_2$ and $a_0 \rightsquigarrow a'_0$. If $U(a) = \epsilon$ we assume $\rho \vdash e \Downarrow w$ for some w and so proceeds as when $U(a) = \mathtt{T}$. So $\models^{\rho}_{denv} e : !^{\mathtt{T}}\ U(t)$ and hence $\rho \vdash e \Downarrow w$ for some w and whenever $\models^{\rho'}_{denv} e' : !^{\mathtt{T}}\ U(t_1)$ we also have $\models^{\rho'}_{denv} w\ e' : !^{U(a_0)}\ U(t_2)$. By the induction hypothesis and the previous fact this implies $\models^{\rho'}_{denv} w\ e' : !^{U(a'_0)}\ U(t'_2)$. Since $U(t_1) = U(t'_1)$ this then establishes the desired $\models^{\rho}_{denv} e : !^{\mathtt{T}}\ U(t')$. □

The second conjunct is immediate from the induction hypothesis. □

Proof of Lemma 10 *(Soundness of matching)*

We proceed by induction on the syntax of the tuple of patterns; we shall write $\vec{p}$ for a tuple of patterns as well as proper patterns.

When $\vec{p}$ is a variable we have $tenv = [v \mapsto t]$ and $\rho = [v \mapsto w]$ where $\exists s : denv \vdash t : s$ and $\models^{[\,]}_{denv} w : !^{\mathtt{T}}\ U(t)$. Clearly $\models_{denv} \rho : U(tenv)$ follows.

When $\vec{p}$ is a constructor c applied to patterns $p_1, \ldots, p_m$ we have

$$tenv = [\,]\ tenv_1 \ldots tenv_m$$

$$\rho = [\,]\ \rho_1 \ldots \rho_m$$

where $denv \vdash p_i : t'_i \Rightarrow tenv_i$ and $p_i : w_i \rightsquigarrow \rho_i$ and

$$denv \vdash denv(c) \geq t'_1 \underset{\mathtt{T}}{\rightarrow} \cdots \underset{\mathtt{T}}{\rightarrow} t'_m \underset{\mathtt{T}}{\rightarrow} t$$

Since U covers t it also covers $t'_1, \ldots, t'_m$. From $\models^{[\,]}_{denv} w : !^{\mathtt{T}}\ U(t)$ we then get $w = c\ w_1 \ldots w_m$ such that $\models^{[\,]}_{denv} w_i : !^{\mathtt{T}}\ U(t'_i)$. The induction hypothesis is now applicable and yields $\models \rho_i : U(tenv_i)$ from which$\models \rho : U(tenv)$ follows.

When $\vec{p}$ is a tuple of patterns $p_1, \ldots, p_m$ we have

$$tenv = [\,]\ tenv_1 \ldots tenv_m$$

$$\rho = [\,]\ \rho_1 \ldots \rho_m$$

where $denv \vdash p_i : t_i \Rightarrow tenv_i$ and $p_i : w_i \rightsquigarrow \rho_i$ and $\models^{[\,]}_{denv} w_i : !^{\mathtt{T}}\ U(t_i)$. The induction hypothesis is applicable and yields $\models \rho_i : U(tenv_i)$ from which $\models \rho : U(tenv)$ follows. □

Proof of Lemma 11 *(Soundness of exhaustiveness)*

We define a well-founded order $\prec$ by

$$(w_1, \ldots, w_m) \prec (w'_1, \ldots, w'_n)$$

iff the sum of the sizes of $w_1, \ldots, w_m$ is strictly less than the sum of the sizes of $w'_1, \ldots, w'_n$. We then proceed by induction on $(w_1, \ldots, w_m)$ ordered by $\prec$ and prove the result by contradiction. To this end assume that

$$\forall i : (p_{i1}, \ldots, p_{im}) : (w_1, \ldots, w_m) \not\hookrightarrow \qquad (\star)$$

We consider each of the three defining clauses for exh in turn. The first clause has $m = 0$ and the desired contradiction is immediate. The last clause has $exh\ (\ldots) = false$ and the desired contradiction is immediate. In the second clause we have two cases. One is when all of $p_{11}, \ldots, p_{n1}$ are variables. Then we also have

$$exh^{denv}_{t_2,\ldots,t_m}([[p_{12}, \ldots, p_{1m}], \ldots, [p_{n2}, \ldots, p_{nm}]])$$
$$\forall i : (p_{i2}, \ldots, p_{im}) : (w_2, \ldots, w_m) \not\hookrightarrow$$

and the desired contradiction then follows from $(w_2, \ldots, w_m) \prec (w_1, \ldots, w_m)$ and the induction hypothesis.

The other case is when there are constructors among $p_{11}, \ldots, p_{n1}$. Then w must be of the form $c\ w'_1 \ldots w'_k$ where

$$denv \vdash denv(c) \geq t'_1 \underset{\mathsf{T}}{\rightharpoondown} \cdots \underset{\mathsf{T}}{\rightharpoondown} t'_k \underset{\mathsf{T}}{\rightharpoondown} t_1$$

and from $\models^{[\,]}_{denv} w : !^{\mathsf{T}}\ U(t_1)$ we have

$$\forall i : \models^{[\,]}_{denv} w'_i : !^{\mathsf{T}}\ U(t'_i)$$

Now classify each p_{i1} into one of three categories:

A: if p_{i1} is a variable then $(\star)$ is equivalent to
$(x_1, \ldots, x_k, p_{i2}, \ldots, p_{im}) : (w'_1, \ldots, w'_k, w_2, \ldots, w_m) \not\hookrightarrow$

B: if p_{i1} is a pattern $c\ p_{i11} \ldots p_{i1k}$ then $(\star)$ is equivalent to
$(p_{i11}, \ldots, p_{i1k}, p_{i2}, \ldots, p_{im}) : (w'_1, \ldots, w'_k, w_2, \ldots, w_m) \not\hookrightarrow$

C: if p_{i1} is a pattern $c'\ p_{i11} \ldots p_{i1k'}$ with $c \neq c'$ then $(\star)$ is equivalent to $true$.

For c as given by $w = c\ w'_1 \ldots w'_k$ next write

$$\vec{p_c} = get\ c\ [[p_{11}, \ldots, p_{1m}], \ldots, [p_{n1}, \ldots, p_{nm}]] = ppp_1 +\!+ \ldots +\!+\ ppp_n$$

where each ppp_i is either $[\,]$, corresponding to p_{i1} of category C, or is of the form $[[x_1, \ldots, x_k, p_{i2}, \ldots, p_{im}]]$, corresponding to p_{i1} of category A, or is of the form $[[p_{i11}, \ldots, p_{i1k}, p_{i2}, \ldots, p_{im}]]$, corresponding to p_{i1} of category of B.

Our assumptions then imply

$$exh^{denv}_{t'_1,\ldots,t'_k,\ t_2,\ldots,t_m}(\vec{pc})$$
$$\forall i : \models^{[\,]}_{denv} w'_i : !^{\mathsf{T}}\ U(t'_i) \wedge \forall i > 1 : \models^{[\,]}_{denv} w_i : !^{\mathsf{T}}\ U(t_i)$$
$$\forall [p'_i, \ldots, p'_k, \ldots, p'_{m+k-1}] \text{ in } \vec{pc} : (p'_1, \ldots, p'_k, \ldots, p'_{m+k-1}) : (w'_1, \ldots, w'_k, w_2, \ldots, w_m) \not\rightarrow$$

and by the induction hypothesis the desired contradiction follows. □

Proof of Lemma 12 *(Monotonicity)*

We proceed by induction on the inference tree for $\rho_1 \vdash e_1 \Downarrow w_1$; this amounts to proceeding by cases on which axiom or rule of Figure 4 with later amendments that has been applied last.

The case $[v]$ is a direct consequence of the assumptions.

The case $[c]$ is trivial.

The case $[IF_T]$. Here $e_i = \texttt{IF}\ e_{i1}\ \texttt{THEN}\ e_{i2}\ \texttt{ELSE}\ e_{i3}$ and $\rho_1 \vdash e_{11} \Downarrow \texttt{True}$. From the induction hypothesis we get w_{21} such that $\rho_2 \vdash e_{21} \Downarrow w_{21}$ and $\texttt{True} \sqsubseteq w_{21}$; this boils down to $w_{21} = \texttt{True}$ so that $\rho_2 \vdash e_{21} \Downarrow \texttt{True}$. Applying the induction hypothesis to $\rho_1 \vdash e_{12} \Downarrow w_1$ we then get w_2 such that $\rho_2 \vdash e_{22} \Downarrow w_2$ and $w_1 \sqsubseteq w_2$; this then proves the result.

The case $[IF_F]$ is similar.

The case $[APP_C]$ is a simple consequence of the induction hypothesis (or see $[APP^{=}_{FUN}]$ below).

The case $[APP^{<}_{FUN}]$ is similar.

The case $[APP^{=}_{FUN}]$. Here $e_i = e_{i1}\ e_{i2}$ and we have

$$\rho_1 \vdash e_{11} \Downarrow (\texttt{FUN}\ \overline{\rho}_1\ v\ \vec{p_1} \Rightarrow \overline{e}_{11} \ldots)\ w_{11} \ldots w_{1(m-1)}$$
$$\rho_1 \vdash e_{12} \Downarrow w_{1m}$$
$$\overline{\rho}_1[v \mapsto \texttt{FUN}\ \overline{\rho}_1\ v\ \vec{p_1} \Rightarrow \overline{e}_{11} \ldots] \vdash (\lambda \vec{p_1}.\overline{e}_{11} \ldots)\ w_{11} \ldots w_{1m} \Downarrow w_1$$

Applying the induction hypothesis to these inferences we obtain

$$\rho_2 \vdash e_{21} \Downarrow (\texttt{FUN}\ \overline{\rho}_2\ v\ \vec{p_1} \Rightarrow \overline{e}_{21} \ldots)\ w_{21} \ldots w_{2(m-1)}$$

where $\overline{\rho}_1 \sqsubseteq \overline{\rho}_2$ and $\overline{e}_{1j} \sqsubseteq \overline{e}_{2j}$ and $w_{1j} \sqsubseteq w_{2j}$. Next we obtain

$$\rho_2 \vdash e_{22} \Downarrow w_{2m}$$

where $w_{1m} \sqsubseteq w_{2m}$. Finally

$$\overline{\rho}_2[v \mapsto \texttt{FUN}\ \overline{\rho}_2\ v\ \vec{p_1} \Rightarrow \overline{e}_{21} \ldots] \vdash (\lambda \vec{p_1}.\overline{e}_{21} \ldots)\ w_{21} \ldots w_{2m} \Downarrow w_2$$

where $w_1 \sqsubseteq w_2$. This then proves the desired result.

The case $[\lambda_1]$. This follows from the induction hypothesis and

Fact 23 If $\vec{p} : \vec{w} \rightsquigarrow \rho$ and $\vec{w} \sqsubseteq \vec{w}'$ then there exists ρ' such that $\vec{p} : \vec{w}' \rightsquigarrow \rho'$ and $\rho \sqsubseteq \rho'$. □

The case $[\lambda_2]$. This follows from the induction hypothesis and

Fact 24 If $\vec{p} : \vec{w} \not\rightsquigarrow$ and $\vec{w} \sqsubseteq \vec{w}'$ then $\vec{p} : \vec{w}' \not\rightsquigarrow$. □

The case $[w]$ is trivial.

The cases $[w]', [APP^{<}_{FUN}]'$ and $[APP^{=}_{FUN}]'$ are similar to their non-primed counterparts.

The cases $[w]'', [APP^{<}_{FUN}]'', [APP^{=}_{FUN}]''_1$ and $[APP^{=}_{FUN}]''_2$ do not arise given the assumptions about purity. □

Proof of Lemma 13 *(Continuity)*

We proceed by induction on the inference tree for $\rho \vdash e \Downarrow w$; again this amounts to proceeding by cases on which axiom or rule of Figure 4 with later amendments that has been applied last.

The case $[v]$ is a direct consequence of the assumptions.

The case $[c]$ is trivial.

The case $[IF_T]$. Here $e = \texttt{IF}\ e_1\ \texttt{THEN}\ e_2\ \texttt{ELSE}\ e_3$ and $\rho \vdash e_1 \Downarrow \texttt{True}$ and $\rho \vdash e_2 \Downarrow w$. Applying the induction hypothesis to e_1 we get $\rho' \vdash e'_1 \Downarrow w'_1$ where w'_1 is a labelling of True; this boils down to $w'_1 = \texttt{True}$ so that $\rho \vdash e'_1 \Downarrow \texttt{True}$. Applying the induction hypothesis to e_2 we note that a labelling that is safe for m plus the size of $\rho \vdash e \Downarrow w$ is also safe for m plus the size of $\rho \vdash e_2 \Downarrow w$ and we obtain $\rho' \vdash e'_2 \Downarrow w'$ where w' is a labelling of w that is safe for m. This then proves the desired result.

The case $[IF_F]$ is similar.

The case $[APP_C]$ is a simple consequence of the induction hypothesis (or see $[APP^{=}_{FUN}]$ below).

The case $[APP^{<}_{FUN}]$ is similar.

The case $[APP^{=}_{FUN}]$. Here $e = e_1\ e_2$ and we have

$$\rho \vdash e_1 \Downarrow (\texttt{FUN}\ \overline{\rho}\ v\ \vec{p_1} \Rightarrow \overline{e}_1 \ldots)\ w_1 \ldots w_{m-1}$$

$$\rho \vdash e_2 \Downarrow w_m$$

$$\overline{\rho}[v \mapsto \texttt{FUN}\ \overline{\rho}\ v\ \vec{p_1} \Rightarrow \overline{e}_1 \ldots] \vdash (\lambda \vec{p_1}.\overline{e}_1 \ldots)\ w_1 \ldots w_m \Downarrow w$$

Write m_1, m_2 and m_3 for the sizes of the these trees and note that $m_1 + m_2 + m_3$ is less than the size of $\rho \vdash e \Downarrow w$. Applying the induction hypothesis we obtain (writing $\texttt{FUN}_*$ for FUN or $\texttt{FUN}_l$ for some $l \geq 0$)

$$\rho' \vdash e'_1 \Downarrow (\texttt{FUN}_*\ \overline{\rho}'\ v\ \vec{p_1} \Rightarrow \overline{e}'_1 \ldots)\ w'_1 \ldots w'_{m-1}$$

where the righthand side is safe for $m + m_2 + m_3$ and hence $m + m_3$. Next we obtain

$$\rho' \vdash e_2' \Downarrow w_m'$$

where w_m' is safe for $m + m_1 + m_3$ and hence $m + m_3$. Finally, we obtain

$$\overline{\rho}'[v \mapsto \texttt{FUN}_\star\ \overline{\rho}'\ v\ \vec{p_1} \Rightarrow \ldots] \vdash (\lambda \vec{p_1}.\overline{e}_1' \ldots)\ w_1' \ldots w_m' \Downarrow w'$$

where w' is a labelling of w safe for m. This then proves the desired result.

The case $[\lambda_1]$. This follows from the induction hypothesis and

Fact 25 If $\vec{p} : \vec{w} \rightsquigarrow \rho$ and $\vec{w}'$ is a labelling of $\vec{w}$ safe for m then there exists ρ' such that $\vec{p} : \vec{w}' \rightsquigarrow \rho'$ and ρ' is a labelling of ρ safe for m. □

The case $[\lambda_2]$. This follows from the induction hypothesis and

Fact 26 If $\vec{p} : \vec{w} \not\rightsquigarrow$ and $\vec{w}'$ is a labelling of $\vec{w}$ then $\vec{p} : \vec{w}' \not\rightsquigarrow$. □

The case $[w]$ is trivial.

The cases $[w]', [APP^{<}_{FUN}]'$ and $[APP^{=}_{FUN}]'$ are similar to their non-primed counterparts.

The cases $[w]'', [APP^{<}_{FUN}]'', [APP^{=}_{FUN}]''_1$ and $[APP^{=}_{FUN}]''_2$ do not arise given the assumptions about purity. □

Proof of Lemma 15 *(Inclusiveness)*

We proceed by induction on (t, a) ordered lexicographically and using the continuity of *sem*.

The case $(t_1 \ldots t_n\ tc,\ \texttt{T})$. From $\models^{\rho_k}_{denv} e_k : !^a\ t$ we get w_k such that $\rho_k \vdash e_k \Downarrow w_k$ and $\models w_k : t_1 \ldots t_n\ tc$. Since $(p_k, e_k)_k$ is a chain and *sem* is continuous also $(w_k)_k$ is a chain and its limit w satisfies $\rho \vdash e \Downarrow w$. Given that $t_1 \ldots t_n\ tc$ is *denv*-well-formed the chain $(w_k)_k$ must be a constant chain and hence each element equals w. (This may be proved by structural induction on w.) This proves the desired $\models^{\rho}_{denv} e : !^a\ t$.

The case $(t_1 \underset{a_0}{\rightarrow} t_2,\ \texttt{T})$. From $\models^{\rho_k}_{denv} e_k : !^a\ t$ we get w_k such that $\rho_k \vdash e_k \Downarrow w_k$ and

$$\models^{\rho'}_{denv} e' : !^{\texttt{T}}\ t_1 \text{ implies } \models^{\rho'}_{denv} w_k\ e' : !^{a_0}\ t_2$$

Since $(\rho_k, e_k)_k$ is a chain and *sem* is continuous also $(w_k)_k$ is a chain and its limit w satisfies $\rho \vdash e \Downarrow w$. The induction hypothesis then gives

$$\models^{\rho'}_{denv} e' : !^{\texttt{T}}\ t_1 \text{ implies } \models^{\rho'}_{denv} w\ e' : !^{a_o}\ t_2$$

and this shows the desired $\models^{\rho}_{denv} e : !^a\ t$.

The case (t, ϵ). We assume $\rho \vdash e \Downarrow w$ as otherwise the result is immediate. By continuity of *sem* there must be a cofinal chain of $(\rho_k, e_k)_k$ for which $\rho_k \vdash e_k \Downarrow w_k$. For this cofinal chain we have $\models^{\rho_k}_{denv} e_k : !^{\texttt{T}}\ t$ and the desired $\models^{\rho}_{denv} e : !^{\texttt{T}}\ t$ follows from the induction hypothesis. □

Proof of Lemma 18 *(Soundness of blocks)*

We proceed by induction on the inference tree. (We do not let U and ρ be given a priori for all cases because U must be varied in case $[GEN]$.)

The case $[\epsilon]$. Let U be given and consider pure ρ such that $\models_{denv} \rho : U(tenv)$. Since $tenv' = tenv$ and $\rho \vdash \epsilon \Downarrow \rho$ the result is immediate.

The case $[\,;]$. Let U be given and consider pure ρ such that $\models_{denv} \rho : U(tenv)$. From the induction hypothesis applied to $block_1$ we get $\rho \vdash block_1 \Downarrow \rho_1$ and $denv,\ tenv \vdash block_1 \Rightarrow tenv_1$ such that $\models_{denv} \rho_1 : U(tenv_1)$ and ρ_1 is pure. From the induction hypothesis applied to $block_2$ we get $\rho_1 \vdash block_2 \Downarrow \rho'$ and $denv,\ tenv_1 \vdash block_2 \Rightarrow tenv'$ such that $\models_{denv} \rho' : U(tenv')$ and ρ' is pure. This then proves the desired result.

The case $[FUN_\pi]$. Let π be a permutation and write

$$\bar{t} = t_1 \underset{\mathtt{T}}{\rightarrow} \ \dots\ \underset{\mathtt{T}}{\rightarrow} t_m \underset{a}{\rightarrow} t$$

Let U be a ground and $denv$-well-formed substitution that covers $tenv' = tenv[v \mapsto \bar{t}]$ and let pure ρ be given such that $\models_{denv} \rho : U(tenv)$. Define

$$w_0 = \mathtt{FUN}\ \rho\ v\ \vec{p_1} \Rightarrow e_1\ \&\ \dots\ \&\ v\ \vec{p_n} \Rightarrow e_n$$

$$\rho' = \rho[v \mapsto w_0]$$

and note that $\rho \vdash block \Downarrow \rho'$. The desired result $\models_{denv} \rho' : U(tenv[v \mapsto \bar{t}])$ then boils down to

$$\models^{[\,]}_{denv} w_0 : !^{\mathtt{T}}\ U(\bar{t})$$

(using a few simple facts about validity). First we observe that $[\,] \vdash w_0 \Downarrow w_0$. The next step is formally by induction on m. It boils down to assuming

$$\models^{\rho_i}_{denv} e_i' : !^{\mathtt{T}}\ U(t_i)$$

(for pure ρ_i and e_i') and defining (necessarily pure) w_i by

$$\rho_i \vdash e_i' \Downarrow w_i$$

and showing

$$\models^{[\,]}_{denv} w_0\ w_1 \dots w_m : !^{U(a)}\ U(t)$$

where we have used the Corollary 8 on Semantic Equivalence. We now proceed by cases on the value of $U(a)$.

The subcase $U(a) = \epsilon$. We shall write

$$w_0^{[k]} = \mathtt{FUN}_k\ \rho\ v\ \vec{p_1} \Rightarrow e_1\ \&\ \dots\ \&\ v\ \vec{p_n} \Rightarrow e_n$$

and by the Lemma 15 on Inclusiveness it suffices to prove

$$\models^{[\,]}_{denv} w_0^{[k]} w_1 \ldots w_m : !^{\epsilon}\; U(t)$$

by induction on k. When $k = 0$ this is immediate since $w_0^{[0]} w_1 \ldots w_m$ does not evaluate. For $k > 0$ we assume

$$[\,] \vdash w_0^{[k]} w_1 \ldots w_m \Downarrow w$$

and our proof obligation now amounts to

$$\models^{[\,]}_{denv} w : !^{U(a)}\; U(t)$$

using the Corollary 8 on Semantic Equivalence. The last rule used must be $[APP^{=}_{FUN}]'$ and so

$$\rho[v \mapsto w_0^{[k-1]}] \vdash (\lambda \vec{p_1}.e_1 \;\&\; \ldots \;\&\; \vec{p_n}.e_n)\; w_1 \ldots w_m \Downarrow w$$

The last $[\lambda_1]$ rule used identifies an index i such that

$$\vec{p_i} : (w_1, \ldots, w_m) \rightsquigarrow \rho_i$$
$$\rho[v \mapsto w_0^{[k-1]}]\rho_i \vdash e_i \Downarrow w$$

The Lemma 10 on Soundness of matching gives

$$\models_{denv} \rho_i : U(tenv_i)$$

where $denv \vdash \vec{p_i} : (t_1, \ldots, t_m) \Rightarrow tenv_i$. Together with the induction hypothesis of the numerical induction and the assumption on ρ we now have

$$\models_{denv} \rho[v \mapsto w_0^{[k-1]}]\rho_i : U(tenv[v \mapsto \overline{t}]tenv_i)$$

From the Lemma 9 on Soundness of expressions we have

$$denv,\; tenv[v \mapsto \overline{t}]tenv_i, \{v\} \models e_i : !^{a}\; t \;\&\; W_i$$

and this gives

$$\models^{\rho[v \mapsto w_0^{[k-1]}]\rho_i}_{denv} e_i : !^{U(a)}\; U(t)$$

which is equivalent to the desired

$$\models^{[\,]}_{denv} w : !^{U(a)}\; U(t)$$

(using the Corollary 8 on Semantic Equivalence).

The subcase $U(a) = \mathtt{T}$. The partial order $<^{\pi}_{a}$ of $[FUN_\pi]$ gives rise to a well-founded order $<^{\pi}_{U(a)}$ on tuples $(w_1, \ldots, w_m)$ of values because $U(a) = \mathtt{T}$. The desired result follows from

$$(\forall i : \models^{[\,]}_{denv} w_i : !^{\mathtt{T}}\ U(t_i) \wedge w_i \text{ pure }) \Rightarrow \models^{[\,]}_{denv} w_0\ w_1 \ldots w_m : !^{\mathtt{T}}\ U(t)$$

which we prove by induction on this well-founded order. So consider a pure tuple $(w_1, \ldots, w_m)$ such that the result holds for all smaller tuples. From the Lemma 11 on Soundness of exhaustiveness there exists an index i such that

$$\forall j < i : \vec{p_j} : (w_1, \ldots, w_m) \not\rightsquigarrow$$
$$\vec{p_i} : (w_1, \ldots, w_m) \rightsquigarrow \rho_i$$

and consequently

$$\rho \vdash w_0\ w_1 \ldots w_m \Downarrow w \quad \text{ iff } \rho[v \mapsto w_0]\rho_2 \vdash e_i \Downarrow w$$

Now recall the existence of a pure value $w_{U(t)}$ satisfying $\models^{[\,]}_{denv} w_{U(t)} : !^{\mathtt{T}}\ U(t)$ and define the impure $w^{[\,]}_0$ by

$$w^{[\,]}_o = \mathtt{FUN}\ [<^{\pi}_{U(a)}, w_1, \ldots, w_m, w_{U(t)}]\rho\ v\ \vec{p_1} \Rightarrow e_1\ \&\ \ldots\ \&\ v\ \vec{p_n} \Rightarrow e_n$$

and note that by assumption on $(w_1, \ldots, w_m)$ this is a total function satisfying

$$(\forall i : \models^{[\,]}_{denv} w'_i : !^{\mathtt{T}}\ U(t_i)) \Rightarrow \models^{[\,]}_{denv} w^{[\,]}_0\ w'_1 \ldots w'_m : !^{\mathtt{T}}\ U(t)$$

Our assumptions now amount to

$$\models_{denv} \rho : U(tenv)$$
$$\models_{denv} [v \mapsto w^{[\,]}_0] : [v \mapsto U(\overline{t})]$$
$$\models_{denv} \rho_i : U(tenv_i)$$

where $denv \vdash \vec{p_i} : (t_1, \ldots, t_m) \Rightarrow tenv_i$ and we have used the Lemma 10 on Soundness of matching. Using the Lemma 9 on Soundness of expressions we have

$$denv,\ tenv[v \mapsto \overline{t}]tenv_i, \{v\} \models e_i : !^a\ t\ \&\ W_i$$

and this gives

$$\models^{\rho[v \mapsto w^{[\,]}_0]\rho_i}_{denv}\ e_i : !^{U(a)}\ U(t)$$

This means that

$$\rho[v \mapsto w_0^{[\,]}]\rho_i \vdash e_i \Downarrow w \qquad (\star)$$
$$\models^{[\,]}_{denv} w : !^{\mathsf{T}}\ U(t)$$

Now consider an exposed (and maximal) call $v\ e''_1 \dots e''_m$ in $(\star)$. It will have the form

$$\rho'' \vdash v\ e''_1 \dots e''_m \Downarrow w''$$

where

$$\rho'' = \rho[v \mapsto w_0^{[\,]}]\rho_i$$

and we also know

$$(e''_1, \dots, e''_m) <^{\pi}_{U(a)} \vec{p_i}$$

Now define $w''_1, \dots, w''_m$ by

$$\rho'' \vdash e''_j \Downarrow w''_j$$

and note that for $\vec{p_i} = p_{i1} \dots p_{im}$ we have

$$\rho'' \vdash p_{ij} \Downarrow w_j$$

as a consequence of $\vec{p_i} : (w_1, \dots, w_m) \rightsquigarrow \rho_i$ and the definition of ρ''. From

Fact 27 If $\rho \vdash e_1 \Downarrow w_1$ and $\rho \vdash e_2 \Downarrow w_2$ and $e_1 <_a e_2$ then $w_1 <_a w_2$.

Proof The induction hypothesis incorporates a similar claim for $\leq_a$ and proceeds by induction on $e_1 <_a e_2$. □

We then have

$$(w''_1, \dots, w''_m) <^{\pi}_{U(a)} (w_1, \dots, w_m)$$

Hence $w_0^{[\,]}$ is replacable by w_0 in ρ'' and $(\star)$ is replacable by

$$\rho[v \mapsto w_0]\rho_i \vdash e_i \Downarrow w$$

Then the desired

$$\models^{[\,]}_{denv} w_0\ w_1 \dots w_m : !^{\mathsf{T}}\ U(t)$$

follows.

The case $[GEN]$. Let U be given as a ground and $denv$-well-formed substitution that covers $tenv'[v \mapsto \forall tav.ts]$ where $tav \notin FTV(tenv') \cup FAV(tenv')$. In case $tav \in$ dom (U) write U' for the restriction of U that is undefined on tav and note that it still covers $tenv'[v \mapsto \forall tav.ts]$. Let pure ρ be given such that $\models \rho : U(tenv)$ and hence $\models \rho : U'(tenv)$. From the induction hypothesis it follows that whenever V is a ground and $denv$-well-formed substitution with domain $\{tav\}$, we have

$$\rho \vdash block \Downarrow \rho'$$

$$\models_{denv} \rho' : (U' \cup V)(tenv'[v \mapsto ts])$$

Since V is arbitrary this gives

$$\models_{denv} \rho' : U'(tenv[v \mapsto \forall tav.ts])$$

from which the desired result follows. □

Proof of Theorem 19

Write

$$[\texttt{'Bool} : 0, \texttt{True} : \texttt{'Bool}, \texttt{False} : \texttt{'Bool}] \vdash defn \;\Rightarrow denv$$

$$denv, [\,] \vdash block \Rightarrow tenv$$

$$denv,\ tenv,\ \emptyset \vdash e : !^a\ t\ \&\ W$$

for the justification for $\vdash defn\ block\ e : !^a\ t$. Now let U be a ground and $denv$-well-formed substitution that covers a and t. Let V be an extension of U that additionally covers $tenv$. From the Lemma 18 on Soundness of blocks we have

$$[\,] \vdash block \Downarrow \rho$$

$$\models_{denv} \rho : V(tenv)$$

and from the Lemma 9 on Soundness of expressions we have

$$\models^{\rho}_{denv} e : !^{V(a)}\ V(t)$$

which amounts to

$$\models^{\rho}_{denv} e : !^{U(a)}\ U(t)$$

If $U(a) = \texttt{T}$ this means that $\rho \vdash e \Downarrow w$ for some w and hence $\vdash defn\ block\ e \Downarrow w$. When this is the case we have $\models^{[\,]}_{denv} w : !^{\texttt{T}}\ U(t)$ by the Corollary 8 on Semantic equivalence. □

Discussion of construction of algorithm

The key focus of this paper is the formulation of an inference based termination analysis and the techniques needed for a soundness result based on operational semantics. We have added sufficient complexity to our inference system that it is able to tackle a reasonably large class of function definitions, yet we have stopped short of introducing any features that might make the system undecidable. The development of an efficient implementation is likely to involve the construction and solution of constraints and is beyond the scope

of the present paper. However, to substantiate our belief that our inference system is indeed decidable we now *sketch* how the standard type inference algorithm $\mathcal{W}$ [14] may be modified to produce the desired results.

First we would inline rule $[GEN]$ with rule $[FUN_\pi]$. Next note that the treatment of algebraic data types does not go beyond what is done in the adaptation of $\mathcal{W}$ to Standard ML [15]. In a similar vein the need to guess the types t_i for rule $[FUN_\pi]$ does not go beyond the pattern matching found in Standard ML. Also it should be obvious that collecting the set W of maximal calls is purely syntactic and does not interfere with type inference. The only potentially trouble-some ingredients of our inference system are the use of permutations and the annotations on arrows.

The permutations may give rise to inefficiency if functions have many arguments. This does not affect decidability because there is only a finite number of permutations over a given number of arguments (readily determined from the syntax). Hence one can simply cycle through all permutations for one that will allow to declare the function total; only if this fails for all permutations will partiality be declared.

Adding annotations on the arrows of function spaces presents no problems either. This is clear from the many algorithms for type and effect inference that are able to take care of polymorphism (e.g. [25]). The potential problem is with the notion of sub-typing that allows to replace `T` by any other annotation. There are at least two ways to deal with this. One is to adapt the algorithm for sub-effecting presented in [26] to take special care to assume that constraints always match. Another is to borrow the two-stage approach to sub-typing algorithm also presented in [26]; here the underlying polymorphic types are computed first before strictness termination annotations are added.

In conclusion we believe that it is within state-of-the-art to produce an type and totality reconstruction algorithm although the approach based on the above sketch may be ineffecient.

LaTeXed February 16, 1995.