Recursion and Dynamic Data-structures in Bounded Space: Towards Embedded ML Programming

John Hughes Lars Pareto

Department of Computing Science
Chalmers University; S 412 96 Göteborg
{rjmh,pareto}@cs.chalmers.se

Abstract

We present a functional language with a type system such that well typed programs run within stated space-bounds. The language is a strict, first-order variant of ML with constructs for explicit storage management. The type system is a variant of Tofte and Talpin's region inference system to which the notion of sized types, of Hughes, Pareto and Sabry, has been added.

1 Introduction

During the last decade, declarative programming has found its way to large-scale industrial practice: Erlang [AVW93] is used by Ericsson to program telecommunication systems; Lustre [HCRP91] by Aerospatiale (among others) for embedded systems programming. Aimed at different domains, these two languages are different in nature. Erlang is rich, but not very safe: it is a general purpose programming language extended with constructs for concurrency, but does not prevent a program from crashing — it's not even statically typed. Lustre is safe but very restricted: it guarantees that all programs terminate and that they do not run out of space, but then, all computations must be expressed using booleans, numbers, streams and operations on these — not even function definitions are allowed.

These languages might be sufficient for their intended domains, but it is not hard to think of applications where they are not: picture an embedded program that maintains a few lookup-tables. In Lustre, such a computation cannot conveniently be expressed. In Erlang it can, but at the cost of a potential abortion at run-time. That a program eventually computes a result is a fundamental requirement on any embedded program.

We propose a programming language that guarantees termination and execution in constant space and that allows user-defined recursive functions and dynamic data-structures.

Our approach is to use a type and effect system with 1) a built in induction principle to guarantee that recursive programs terminate, with 2) a notion of sizes for data-types to allow manipulation of dynamic data-structures (such as lists) in constant space; that 3) maintains an upper bound on the height of the run-time stack to guarantee that a program never runs out of stack space, and that 4) maintains an upper bound on the memory allocations to guarantee that a program never runs out of heap space. The technique for termination proofs (1) and the notion of sizes (2) are immediately inherited from our previous work on sized types [HPS96, Par98], which does not address the bounded space problem. The techniques for memory allocation (4) are based on Tofte's and Talpin's work on region inference [TT97], which takes storage use into account, but which cannot give space bounds for recursive data-structures. The use of a type and effect system to compute an upper bound for the run-time stack (3) is novel.

In the next section, we will introduce our working language MML (embedded ML): a strict, first order, purely functional language with explicit regions; we start by motivating our approach, and then give an informal introduction to regions and to what programming in the language looks like (sec. 2.1 and 2.2). The technical part of the paper defines the syntax (sec. 2.3), the semantics (sec. 3), and the type system (sec. 4) of MML. The technical part is followed by a discussion of the correctness of the type system (sec. 5.1 and 5.2), of the implementation aspects of MML (sec. 5.3, 5.4, and 5.5), of four possible language extensions (sec. 5.6, 5.7, 5.8, and 5.9), and of two possible widenings of our approach (sec. 5.10, and 5.11).

2 A Functional Language with Explicit Regions

One praised aspect of functional programming is that it entirely relieves the programmer from specifying where in memory any run-time values should reside — storage issues are left to the compiler and to the run-time-system.

Coincidentally, a long cursed aspect of functional programming has been that programs use far too much memory and that its hard for the programmer to mend this — functional compilers and run-time systems seem not able to solve these storage issues in a satisfying way.

A valid question, thus, is whether traditional functional language designs have taken the abstraction from storage management too far? Will there not always be cases where the programmer can do a better job than the compiler, and in which he is willing to undertake this effort? Is a programming language that does not abstract away from storage management necessarily a lesser language?

The ML Kit project at Diku has convincingly demonstrated that SML can be given a semantics that also takes storage management into account, that the programmer who understands this can tune his programs to use significantly less space than before, and that the traditional advantages of high-level programming by no means are lost by giving the programmer control over the storage management [TBE+].

Undeniably, advances in compiler technology will reduce the potential win of this approach, but we shall proceed under the assumption that programmers — at least in the near future — are cleverer than the compilers they use.

In the ML Kit, storage management is implicit. The source language, SML, is automatically translated into a compiler intermediate language with constructs for storage management. It is an understanding of this intermediate language, and of the translation process that gives the programmer the desired control.

Our approach, instead, is to make storage management explicit. The storage constructs of the ML Kit's intermediate language are present at the source level of ours. We believe that in the domain of embedded systems, programmers want that explicit control.

2.1 Programming with Regions

Regions can be thought of as short-lived heaps that are introduced and disposed of by the programmer. A region is empty at its point of creation, successively filled with values during its lifetime, and discarded in one go when disposed. In contrast to heaps, regions are never garbage-collected: neither during their lifetimes, nor when they cease to exist.

Regions are associated with a *region size* that denotes the number of machine words occupied by a region; this size is constant during the lifetime of a region.

In MML, a region is introduced by the construct

$$\textbf{letreg}\ x\#e\ \textbf{in}\ e'$$

which creates a region of the size determined by the expression e (a number), binds a reference to this region (a name) to the variable x, evaluates e' in the current environment extended with this binding, and which, finally, discards the region before returning the value of e'. Clearly, the value of e' must not be stored in this region; the region is merely to be used by the computations that produce this value.

Two kinds of values — tuples and constructors — can be stored in regions, and these values can only be stored in regions. To specify in which region a value should be stored tuple- and constructor-expressions are augmented with region parameters. Atomic values, such as numbers, are stored either on the stack or in the region of an enclosing tuple or constructor, and thus do not need any region parameters. For example, the program to store the pair $(5, 66)$ in a region accessible by x, and to compute the sum of the elements of this pair (which can be retrieved with the projection operators π_1 and π_2) is

$$\textbf{letreg}\ x\#\check{2}\ \textbf{in}\ \ \textbf{let}\ a\ =\ (5,66)_x\ \textbf{in}\ \pi_1\ a + \pi_2\ a$$

The annotation on the region size indicates that the expression is a *region size expression* and not an ordinary natural number. This distinction is necessary, because these two classes of numbers have different types — we shall return to this below.

The size of the region above is taken to be 2 because two words are necessary to store a pair.

Regions in MML can be passed to functions and also be returned. The above program, for example, can be rewritten to:

$$\begin{aligned} f\ x\ y\ r\ &=\ (x,y)_r \\ g\ p\ &=\ \pi_1\ p + \pi_2\ p \end{aligned}$$
$$\textbf{letreg}\ x\#\check{2}\ \textbf{in}\ \ \textbf{let}\ a\ =\ f\ 5\ 66\ x\ \textbf{in}\ g\ a$$

Here, the function f creates a pair in an arbitrary region specified by the region parameter r; the function g computes the sum of a pair stored in an arbitrary region — note that no region parameter is necessary for this.

Regions cannot, however, be constructed, decomposed or compared: the only way to introduce a region is by the letreg-construct; the only way to use a region (except for passing it around) is as a storage designator in a tuple- or a constructor expression.

2.2 Sized Region Types

Consider the program

$$\begin{aligned} &\mathit{len}\ \mathit{xs}\ =\ \textbf{case}\ \mathit{xs}\ \textbf{of}\ \mathsf{nil} \to 0;\ \mathsf{cons}\ x\ \mathit{xs}' \to 1 + \mathit{len}\ \mathit{xs} \\ &\mathit{fall}\ n\ r\ =\ \mathsf{cons}\ n\ (\ \textbf{case}\ n\ \textbf{of} \\ &\qquad\qquad\qquad\qquad 0 \to \mathsf{nil}\ r \\ &\qquad\qquad\qquad\qquad x+1 \to \mathit{fall}\ x\ r)\ r \\ &\textbf{letreg}\ \ x\#\check{4}\check{*}\check{3}\check{+}\check{1}\ \textbf{in}\ \mathit{len}(\mathit{fall}\ 4\ x) \end{aligned}$$

which presents two typical programming slips, that would both lead to a crash. In the definition of *len*, the recursive call is given xs rather than xs', causing the run-time stack to grow out of hand; in the letreg-declaration, the programmer has correctly remembered that it takes 3 words to store a cons cell, and one word to store nil; what he has overlooked is that the list returned by *fall* $4\ x$ — or not returned in this case — has a length of five rather than four. This leads to an overfull region.

Let us look at how the type system of MML prevents errors of these kinds. We start by looking at the basic notion of sized types. First of all, every inductively defined value is associated with a size that denotes the number of iterations of the type definition necessary to define the value, e.g., the natural number 1 (which denotes succ zero) has size two; the list nil has size one. No value can have size zero. Moreover, every inductively defined type carries a size that gives an upper bound on the sizes of the values in the type, e.g., the type $\mathsf{List}_3\ \mathsf{Nat}_2$ contains the elements $\mathsf{nil}, 0:\mathsf{nil}, 1:\mathsf{nil}, 0:0:\mathsf{nil}, \ldots, 1:1:\mathsf{nil}$, but neither $2:\mathsf{nil}$, nor $0:0:0:\mathsf{nil}$. (Refer to [HPS96, Par98] for more details on sized types.)

This paper extends these types with region parameters: tuple- and data-types (but, notably, not numbers) are augmented with a parameter that specifies in which region a value is stored.

The type of the intended list length function above, for example, would typically be

$$\mathit{len} :: \forall\,\mathsf{k}\,\mathsf{a}\,\mathsf{r}\,.\,\mathsf{List}_\mathsf{k}\ \mathsf{a}\ \mathsf{r} \to \mathsf{Nat}_\mathsf{k}\ \textbf{with}\ \delta = 5\mathsf{k}$$

and with the following interpretation: given a list of size k, with elements of type a, and that resides in region r, *len* will return a natural number of size k.

Another extension present in this type is the *stack effect* δ, which states that the temporary use of stack-space during any execution of *len*, is 5 times k words. (Five, because, at each recursive call, the values of x, xs', 1, and xs' again, as well as a return address, will reside on the run-time stack — this will be evident when we introduce the abstract machine semantics below.)

The termination-proof-technique employed by the type system, and that traps the first bug above, consists of a base case, and an inductive case, which, for *len*, reduces to showing that

$$len''\ xs\ =\ \textbf{case}\ xs\ \textbf{of}\ \mathsf{nil} \to 0;\ \mathsf{cons}\ x\ xs' \to 1 + len'\ xs$$

is typable under the type assignment

$$len'\ ::\ \mathsf{List_k\ a\ r} \to \mathsf{Nat_k}\ \textbf{with}\ \delta = 5\mathsf{k}$$
$$len''\ ::\ \mathsf{List_{k+1}\ a\ r} \to \mathsf{Nat_{k+1}}\ \textbf{with}\ \delta = 5 * (\mathsf{k}+1)$$

But this is not possible. The variable xs cannot both have size k and size $\mathsf{k}+1$, which this non-proof would require. The program is rejected on this ground.

If we correct the bug by replacing the second occurrence of xs, with xs', this verification step will succeed.

To see how the second bug is detected by the type system, we introduce a type for *fall*

$$fall :: \forall \mathsf{kr}\,.\, \mathsf{Nat_k} \times \mathsf{r} \to \mathsf{List_{k+1}}\,(\mathsf{Nat_k})\ \mathsf{r}\ \textbf{with}\ \delta = 5\mathsf{k};\ \mathsf{r} \mathrel{+}= 3\mathsf{k}+1$$

which causes no problems. By coincidence, the stack effect is again $5\mathsf{k}$: at each recursive call, the values of n, x, x, r, plus a return address will have been pushed onto the stack. The type of *fall* also has a *put effect* ($\mathsf{r} \mathrel{+}= 3\mathsf{k}+1$) telling us that the function will store at most $3\mathsf{k}+1$ words in region r. (Three, because every cons-node occupies one word each for the constructor-name, the head, and for the tail.)

Returning to our fallacious program, let ρ be the type of the region bound to x by the letreg-construct, and note that 4 has type $\mathsf{Nat_5}$, the type of *fall* gives that the expression $fall\ 4\ x$ has type $\mathsf{List_6\ Nat_5}\ \rho$ with the stack effect $\delta = 28$ (five times five, plus three words for the parameters and the return address of the call) and the put effect $\rho \mathrel{+}= 16$. These effects will be propagated through *len*, so the body of the letreg-expression will also have the put effect $\rho \mathrel{+}= 16$. The type system rejects the program because $16 \not\leq 4*3+1$. To fix the problem, the programmer simply has to increase the region size by three.

The type of the corrected program, is

$$\mathsf{Nat_5}\ \textbf{with}\ \delta = 33;\ \phi = 17$$

Notice that this type has no put effects: put-effects concern regions in scope, and at the top-level of a program, there are no regions in scope. Instead, the program has a *store effect* ϕ that gives the maximum size of the store during its evaluation.

Notice, also, that this store-effect is one word larger than the put-effect of the let-expression's body. To understand why, we need to take a look at the run-time system.

The run-time system of MML maintains two stacks: a value stack in the style of traditional implementations of strict languages, and a region stack in style of the ML Kit, and on which letreg-expressions will push and pop *region frames* — run-time representations of regions. A region of size s is represented by s machine words in sequence preceded by a region offset that divides the region into an *occupied* and a *free* part. This offset will be 0 at the creation of a region, and successively increased by the allocations in that region by the number of words required to store a value. The offset is never decreased. For example, writing $w \cdots w$ for a sequence of machine words, and _ for an undefined value, the region stack during the evaluation of

$$\begin{array}{l}\textbf{letreg}\ \ x\#\check{4}\ \textbf{in letreg}\ \ y\#\check{2}\ \textbf{in}\\ \textbf{let}\ a = (7,8,9)_x\ \textbf{in let}\ b = (6,7)_y\ \textbf{in}\ 0\end{array}$$

would be 0 _ _ _ _ just after the binding of x, and 3 7 8 9 _ 2 6 7 after the binding of b. This representation explains why the store effect of a letreg-expression is that of its body's put-effect plus one.

As for the stack effect, we need one word to store the region on the stack. Thus the stack effect of a letreg-expression is one greater than that of its body. In this example, the call of *len* requires 32 words (30 plus two for the parameter and the return address of the first call). Hence the stack effect $\delta = 33$.

As mentioned above, region size expressions are not ordinary natural numbers (hence the annotations). The distinction is required by the type system, which, at letregs, needs to know the region size exactly. As the type $\mathsf{Nat_k}$ only give bounds on sizes, we need a different type, $\mathsf{Size_k}$, that gives us this. To avoid overloading, we use a different syntax for region size expressions and write $\check{n}$, $e \mathbin{\check{+}} e$, etc., as in

$$\check{1}\ ::\ \mathsf{Size_1} \qquad \check{1}\check{+}\check{2}\ ::\ \mathsf{Size_3}$$

Types with this precision will, of course, impose severe restrictions on the programs that compute with their elements; one cannot, for example, use a case expressions to return different sizes. Fortunately, such computations do not have to be very expressive. What we typically want to express is how the put effect of a function depends on the arguments of this function, i.e, the dependency that also appears in the type of a function. Now, these dependencies are themselves severely restricted: as we shall see, size expressions are confined to linear functions. As long as we can express such, programming with sizes is permissive enough.

Let us illustrate the use of computations with sizes by showing how our main example can be made more readable through the use of a function $fall'$ to compute the put effect of *fall*:

$$\begin{array}{l}len :: \forall \mathsf{kar}\,.\, \mathsf{List_k\ a\ r} \to \mathsf{Nat_k}\ \textbf{with}\ \delta = 5k\\ len\ xs\ = \textbf{case}\ xs\ \textbf{of}\ \mathsf{nil} \to 0;\ \mathsf{cons}\ x\ s \to 1 + len\ s\end{array}$$

$$\begin{array}{l}fall :: \forall \mathsf{kr}\,.\, \mathsf{Nat_k} \times \mathsf{r} \to \mathsf{List_{k+1}}\,(\mathsf{Nat_k})\ \mathsf{r}\ \textbf{with}\\ \qquad\qquad\qquad\qquad \delta = 5k;\ r \mathrel{+}= 3k+1\\ fall\ n\ r\ =\ \mathsf{cons}\ n\ (\ \textbf{case}\ n\ \textbf{of}\\ \qquad\qquad\qquad 0 \to \mathsf{nil}\ r\\ \qquad\qquad\qquad x+1 \to fall\ x\ r)\ r\end{array}$$

$$\begin{array}{l}fall'\ ::\ \forall \mathsf{k}\,.\,\mathsf{Size_k} \to \mathsf{Size_{3k+1}}\\ fall'\ s\ =\ \check{3}\check{*}s \mathbin{\check{+}} \check{1}\end{array}$$

$$\textbf{letreg}\ \ x\#fall'\ \check{5}\ \textbf{in}\ len(fall\ 4\ x)$$

2.3 The Syntax of MML

We move on to formalise our development so far, and start with MML's syntax:

$$
\begin{array}{lcl}
P & ::= & g \cdots g\; e \\
g & ::= & d \cdots d \\
d & ::= & f\; x \cdots x \;=\; e \\
e & ::= & n \mid \check{n} \mid x \mid c\; e \cdots e\; e \mid e \diamond e \mid f\; e \cdots e \\
 & \mid & \mathbf{let}\; x = e\; \mathbf{in}\; e \\
 & \mid & \mathbf{letreg}\; x\#e\; \mathbf{in}\; e \mid (e, \ldots, e)_e \mid \pi_n\; e \\
 & \mid & \mathbf{case}\; e\; \mathbf{of}\; 0 \to e \,[\!]\, x+1 \to e \mid \mathbf{case}\; e\; \mathbf{of}\; A \cdots A \\
\diamond & ::= & + \mid - \mid * \mid \check{+} \mid \check{-} \mid \check{*} \\
A & ::= & c\; x \cdots\; x \;\to\; e \\
n & \in & \textsc{Nat} \qquad x \in \textsc{Var} \qquad c \in \textsc{Con} \qquad f \in \textsc{FunVar}
\end{array}
$$

A program (P) consists of a sequence of groups of mutually recursive function definitions (g) followed by an expression. Such groups, in turn, are sequences of basic function definitions (d) that take the form $f\; x \cdots x \;=\; e$. The top-level of a program is the only place where function definitions are allowed: there are no lambda expressions or other means of defining a function within an expression.

The form of an expression (e) is more-or-less standard, but with occasional twists that call for our attention:

Numbers are present both as naturals (n) and as sizes ($\check{n}$) as discussed above. Variables (x) are standard.

Constructor expressions ($c\; e_1\; \cdots\; e_n\; e$) consist of n argument expressions, in the style of Haskell, plus an additional region expression. Constructors cannot be partially applied.

The arithmetic operators ($\diamond$) are restricted to addition, subtraction, and multiplication, but more operators could easily be added.

Function calls ($f\; e_1 \cdots e_n$) are standard. Partial application is not allowed.

Let expressions ($\mathbf{let}\; x = e\; \mathbf{in}\; e'$) are monomorphic, and must be non-recursive (lacking expressions of function type, the language has no need for a polymorphic let).

The letreg- ($\mathbf{letreg}\; x\#e\; \mathbf{in}\; e'$), tupling- ($(e_1, \ldots, e_n)_e$), and projection-constructs ($\pi_n\; e$) have already been covered.

There are case expressions of two forms: for natural numbers ($\mathbf{case}\; e\; \mathbf{of}\; 0 \to e_1 \,[\!]\, x+1 \to e_2$) and for constructor expressions ($\mathbf{case}\; e\; \mathbf{of}\; A_1 \cdots A_n$), that is, for unboxed values (values stored on the run-time stack) and for boxed values (values stored in a region). Constructor alternatives (A) are restricted to simple patterns ($c\; x_1\; \cdots\; x_n$) with one variable for each constructor parameter, but the last one. (Pattern matching cannot be used to retrieve region names.)

The syntax of MML's types,

$$
\begin{array}{lcl}
\sigma & ::= & \forall k \,.\, \sigma \mid \forall t \,.\, \sigma \mid \forall \rho \,.\, \sigma \mid \nu \\
\nu & ::= & \tau \times \cdots \times \tau \overset{\delta;\, p;\, \phi}{\to} \tau \mid \tau \times \cdots \times \tau \times \rho \to \kappa \\
\tau & ::= & t \mid \rho \mid [\tau \times \cdots \times \tau]_\rho \mid \mathsf{Nat}_s \mid \mathsf{Size}_s \mid \kappa \\
\kappa & ::= & T_{s \cdots s}\; \tau \cdots \tau\; \rho \\
p & ::= & \{\rho \mapsto n, \ldots, \rho \mapsto n\} \\
s & ::= & \omega \mid i \\
i & ::= & k \mid n \mid n * i \mid i + i \\
\multicolumn{3}{l}{k \in \textsc{SizeVar} \;\; t \in \textsc{TypeVar} \;\; \rho \in \textsc{RegVar} \;\; \delta, \phi, n \in \textsc{Nat}}
\end{array}
$$

also invites a few remarks. There are three kinds of polymorphism: in size- ($\forall k$), type- ($\forall t$), and region-variables ($\forall \rho$). Polymorphism is restricted to function or constructor types (ν).

Function types ($\tau_1 \times \cdots \times \tau_n \overset{\delta;\, p;\, \phi}{\to} \tau$) take the form of a mapping from a tuple of types rather than from a type; this is to manifest that our language is first order. Function arrows are labelled with the three kinds of effects discussed above: stack effects (δ) which are natural numbers; put effects (p) which are mappings from region variables to natural numbers ($\{\rho_1 \mapsto n_1, \ldots, \rho_n \mapsto n_k\}$); store effects ($\phi$) which are also natural numbers. In the informal development above, we used a sugared syntax for our function types and wrote

$$\tau_1 \times \cdots \times \tau_n \to \tau \;\mathbf{with}\; \delta = n_1;\; \rho \mathrel{+}= n_2;\; \phi = n_3$$

rather than $\tau_1 \times \cdots \times \tau_n \overset{n_1;\, \{\rho \mapsto n_2\};\, n_3}{\to} \tau$.

Constructor types ($\tau_1 \times \cdots \times \tau_n \times \rho \to \tau$) have the same shape as function types, but for the absence of effects. This is because the effects of a constructor application are uniquely determined by its arity and the effects of its arguments.

A basic type (τ) can be:

A standard type variable (t), or a region variable (ρ).

A tuple type ($[\,\tau_1 \times \cdots \times \tau_n\,]_\rho$) composed from element-types, as usual, but here also from a region variable that specifies where the tuple is stored.

One of the built-in types for numbers (Nat_s and Size_s), covered above.

A constructor type (κ). These are introduced by datatype declarations. The general form of a constructor type is $T_{s_1 \cdots s_n}\; \tau_1 \cdots \tau_m\; \rho$, where the first size parameter s_1 gives the size of the type, and where $s_2 \cdots s_n$ and $\tau_1 \cdots \tau_m$ are actual size and type parameters (refer to [Par98] for details); the last parameter ρ plays the same role as in tuple types.

A size expression (s) is either the limit ordinal literal (ω), or a positive integer expression (i). Such an expression, in turn, is either a variable (k), a constant (n), a linear multiplication ($n * i$), or an addition ($i + i$). This form of size expressions is inherited from the theory of sized-types, which requires that: 1) all size expressions are strictly monotonic in all their size variables, and that 2) all size expressions are linear in each size variable [Par98].

The syntax for declarations is:

$$
\begin{array}{lcl}
D & ::= & \forall k \,.\, D \mid \forall t \,.\, D \mid \forall \rho \,.\, D \mid \varepsilon \\
\varepsilon & ::= & \kappa = r \\
r & ::= & c\; \tau \cdots \tau\; \rho \mid \cdots \mid c\; \tau \cdots \tau\; \rho
\end{array}
$$

A declaration (D) is a, possibly quantified, equality (ε) between a constructor type (κ) and a right-hand-type (r); a right hand type is a sequence of tagged type-sequences, each with a trailing region type ($c\; \tau_1 \cdots \tau_n\; \rho$).

We write $\mathbf{D}$ for the set of declarations, and, in this paper, take this set to be

$$\mathbf{D} = \{\, \forall \mathsf{k\,a\,r} \,.\, \mathsf{List_{k+1}\; a\; r} \;=\; \mathsf{nil\; r} \mid \mathsf{cons\; a\; (List_k\; a\; r)\; r} \,\}$$

3 The Semantics of MML

The role of our language semantics is manifold, and we must carefully design the semantics with respect to all its intended uses: the semantics are used to reason about a type system with stack and store effects, so some notion of a stack and a store must be provided; the semantics serve as a specification for compilers, so it must be designed with language implementation techniques in mind; the model is used in numerous proofs, so it must be mathematically tractable.

Under these three requirements, semantics by an abstract-machine in the style of Landin [Lan64] has turned out to be a suitable choice. By adding a store and a few new instructions to his SECD-machine, and by coalescing its dump and stack, we get a semantic model that satisfies these requirements.

3.1 The Abstract Machine

An abstract machine is a transition system between machine states M, which, in MML, take this form:

$$
\begin{array}{lcl}
M & ::= & \langle V, S, E, C \rangle \\
V & ::= & v : V \mid v \\
v & ::= & n \mid \check{n} \mid a \mid \rho \mid c \mid C_{ES} \\
a & ::= & o_\rho \\
S & ::= & \{\rho \mapsto R, \ldots, \rho \mapsto R\} \\
R & ::= & \{o \mapsto v, \ldots, o \mapsto v\}^n \\
E & ::= & \{x \mapsto v, \ldots, x \mapsto v\} \\
C & ::= & \iota : C \mid \mathbf{call}_f \mid \mathbf{ret}_n \mid \mathbf{sret}_n^{C_{ES}} \mid \mathbf{end} \\
\iota & ::= & e \mid \diamond \mid e \,\square_x\, e \mid \not\rho \mid ()^n \mid \pi_n \mid c^n \\
 & \mid & A \cdots A \mid \lambda x.e \mid \{\}_x\, e \\
o & \in & \textsc{Nat} \qquad \rho \in \textsc{RegVar}
\end{array}
$$

A machine state (M) consists of a value stack (V) that represents the run-time stack of a language implementation, a store (S) that represents the region stack, an environment (E) that maps program variables to values, and which has no immediate run-time representation, and an instruction sequence (C) that represents the machine instructions in an executable program.

A value (v) is either a natural number (n), a size ($\check{n}$), an address (a) which points to either a tuple or a constructor in the store, a region identifier (ρ) which can be thought of as a reference to a region, a constructor tag (c), or a continuation C_{ES}, which corresponds to a return address in an activation record.

An address (a) is a pair of the form o_ρ where ρ is a region identifier and o an offset in that region (o is a natural number).

A store (S) is a mapping $\{\rho_1 \mapsto R_1, \ldots, \rho_k \mapsto R_k\}$ from region variables to regions (R), which in turn are fixed-size mappings $\{o_1 \mapsto v_1, \ldots, o_k \mapsto v_k\}^n$ from offsets to values, where k must not exceed n. The value in a store S pointed to by an address o_ρ can thus be expressed as $S(\rho)(o)$. We write $S + \{\rho \mapsto R\}$ to extend a store with region R identified by ρ; to make our manipulation of stores more concise, we introduce the syntactic abbreviations $S(o_\rho) \equiv S(\rho)(o)$ and $S + \{o_\rho \mapsto v\} \equiv S + \{\rho \mapsto S(\rho) + \{o \mapsto v\}\}$.

An instruction sequence (C) is a sequence of ordinary machine instructions (ι) followed by a control flow instruction ($\mathbf{call}_f$, $\mathbf{ret}_n$, $\mathbf{sret}_n^{C_{ES}}$, and $\mathbf{end}$). The execution of an ordinary instruction will remove it from the instruction sequence, change the machine state according to the instruction, and then execute the remainder of the sequence. The execution of a control flow instructions will install a new instruction sequence.

We devote the remainder of this section to the meaning of these instructions.

3.2 Numbers and Variables

All expressions in our language also constitute machine instructions that, when executed, leave one value on the value stack, e.g.,

$$
\begin{array}{lcl}
\langle V, S, E, n : C \rangle & \to & \langle n : V, S, E, C \rangle \\
\langle V, S, E, x : C \rangle & \to & \langle E(x) : V, S, E, C \rangle
\end{array}
$$

3.3 Constructors

The execution of an expression is typically divided into a sequence of sub-computations that, together, form the desired result. To execute a constructor expression, for example, we first execute all argument expressions, which will leave a value for each instruction on the stack; then, we execute an instruction that, using these values, will write the machine representation of the constructor to the store, and that will leave the address to this machine representation on the value-stack:

$$
\begin{array}{lcl}
\langle V, S, E, c\, e_1 \cdots e_n\ e : C \rangle & \to & \langle V, S, E, e_1 : \cdots : e_n : e : c^n : C \rangle \\
\langle \rho : v_n : \cdots : v_1 : V, S, E, c^n : C \rangle & \to & \langle o_\rho : V, S', E, C \rangle;\ o = |S(\rho)_{\mathbf{D}}| \\
 & & S' \equiv S + \{o_\rho \mapsto c\} + \{(o+i)_\rho \mapsto v_i\}
\end{array}
$$

Here, by $S(\rho)_{\mathbf{D}}$ we mean the domain of the region bound to ρ in S, and by $|S(\rho)_{\mathbf{D}}|$ the number of elements in this set: this gives a fresh offset in the designated region (assuming that all binding offsets in $S(\rho)$ have been introduced in this way). A constructor value is simply represented as the constructor tag followed by the argument values. The address left on the value stack will point to the first word of this sequence.

3.4 Arithmetic Operators

The rules for arithmetic operations are straightforward, e.g,

$$
\begin{array}{lcl}
\langle V, S, E, e_1 + e_2 : C \rangle & \to & \langle V, S, E, e_1 : e_2 : + : C \rangle \\
\langle n_2 : n_1 : V, S, E, + : C \rangle & \to & \langle n_1 + n_2 : V, S, E, C \rangle
\end{array}
$$

We omit the rules for ($-$,$*$,$\check{+}$, $\check{-}$, and $\check{*}$).

3.5 Function Calls

At the abstract machine level, a function call can be divided into 1) creation of a continuation to use the computed value, 2) evaluation of the function's arguments, 3) creation of an environment that maps the formal variables of the function to the actual values of the call, 4) execution of the function's body in this environment, 5) removal of actual parameters from the stack, and 6) execution of the continuation. The rules to accomplish this process are:

$$
\begin{array}{l}
\langle V, S, E, f\ e_1 \cdots e_n : C \rangle \ \to\ \langle C_{ES} : V, S, E, e_1 : \cdots : e_n : \mathbf{call}_f \rangle \\
\langle v_n : \cdots : v_1 : C_{ES'} : V, S, E', \mathbf{call}_f \rangle \ \to \\
\qquad\qquad \langle v_n : \cdots : v_1 : C_{ES'} : V, S, \{x_i \mapsto v_i\}, e_f : \mathbf{ret}_n \rangle \\
\langle v : v_n : \cdots : v_1 : C_{ES'} : V, S, E', \mathbf{ret}_n \rangle \ \to\ \langle v : V, S, E, C \rangle;\ S' \subseteq S
\end{array}
$$

When executed, a call-expression-instruction (the first rule) pushes a continuation on the stack, and places the argument expressions plus a call instruction, $\mathbf{call}_f$, in the instruction sequence. The continuation, C_{ES}, consists of the instruction sequence following the call plus the environment and the store at the call; the continuation must carry both the environment and the store, because the environment maps variables to values, and values can be addresses. To capture the store in this way is, of course, not realistic in an implementation, so we play the following trick: we only allow continuations to be reinstalled in machine states with a store larger than that of the continuation (the side condition of the third rule above). In our semantics, this will always be the case so a language implementor can forget about this condition; we need it, however, for our proofs.

The execution of a call-instruction (the second rule) creates the new environment and places the called function's body plus a return instruction in the instruction sequence. To allow the environment to be implemented using indirect addressing relative to the top of the stack (a standard technique), we leave the actual parameters on the stack.

Return instructions (the third rule) are parameterised on the number of arguments that the return-value should be slid over at the return (sliding is another standard technique). In addition to the slid arguments, the return instruction removes the continuation from the stack and reinstalls its instruction sequence and environment.

3.6 Let-Bindings

The execution of a let-expression resembles a function call in many ways; it involves 1) computation of an argument, 2) creation of an environment that binds this value to a variable, 3) creation of a continuation, 4) execution of an expression in the new environment, 5) sliding, and 6) reinstallation of a continuation. The most notable difference is that, in a function call, the continuation is a dynamic object (i.e., it will be present at run-time), whereas, in a let expression, a continuations is a static object (i.e., it will only be present at compile time):

$$\langle V, S, E, \mathbf{let}\, x = e\ \mathbf{in}\ e' : C\rangle \ \rightarrow\ \langle V, S, E, e : \lambda x.e' : C\rangle$$
$$\langle v : V, S, E, \lambda x.e : C\rangle \ \rightarrow\ \langle v : V, S, E + \{x \mapsto v\}, e : \mathbf{sret}_1^{C_{ES}}\rangle$$
$$\langle v : v_n : \cdots : v_1 : V, S, E', \mathbf{sret}_n^{C_{ES'}}\rangle \ \rightarrow\ \langle v : V, S, E, C\rangle;\ \ S' \subseteq S$$

Here, the instruction $\lambda x.e$ provides, "in-line" functions, and the instruction $\mathbf{sret}_n^{C_{ES}}$ static returns. Note that the body of a lambda-instruction is an expression, not an instruction, so nested lambda expressions can never arise.

3.7 Letreg

Letreg expressions, in turn, resemble let-expressions, but there are two important differences: 1) the life and death of a new region is involved in their executions; 2) the expression computed first will not itself be bound to the introduced variable, but used to determine the size of the created region — what is bound is a reference to this region. The rewrite process uses the region-creation instruction $\{\}_x$ and the region-destruction instruction $\not{\phi}$:

$$\langle V, S, E, \mathbf{letreg}\ x\#e\ \mathbf{in}\ e' : C\rangle \rightarrow \langle V, S, E, e : \{\}_x\, e' : C\rangle$$
$$\langle \check{n} : V, S, E, \{\}_x\, e : C\rangle \rightarrow \langle \rho : V, S', E', e : \not{\phi} : \mathbf{sret}_1^{C_{ES}}\rangle;\ \rho \notin S_{\mathbf{D}}$$
$$S' \equiv S + \{\rho \mapsto \{\}^{n+1}\};\qquad E' \equiv E + \{x \mapsto \rho\}$$

$$\langle v : \rho : V, S, E, \not{\phi} : \mathbf{sret}_1^{C_{E'S'}}\rangle \rightarrow \langle v : \rho : V, S \backslash \rho, \{\}, \mathbf{sret}_1^{C_{E'S'}}\rangle$$

By $S_{\mathbf{D}}$ above, we mean the domain of S, and by $S\backslash\rho$ we mean $\{\rho' \mapsto R\ \in\ S \mid \rho' \neq \rho\}$.

3.8 Tuples and Tuple Projections

The tuple construction process is almost identical to that of constructors; the absence of a leading constructor name in the representation is the only difference:

$$\langle V, S, E, (e_1, \ldots, e_n)_e : C\rangle \rightarrow \langle V, S, E, e_1 : \cdots : e_n : e : ()^n : C\rangle$$
$$\langle \rho : v_n : \cdots : v_1 : V, S, E, ()^n : C\rangle \rightarrow \langle o_\rho : V, S', E, C\rangle;\ o = |S(\rho)_{\mathbf{D}}|$$
$$S' \equiv S + \{(o + i - 1)_\rho \mapsto v_i\}$$

Given this representation, tuple projection is trivial:

$$\langle V, S, E, \pi_i\ e : C\rangle \qquad \rightarrow\ \langle V, S, E, e : \pi_i : C\rangle$$
$$\langle o_\rho : V, S, E, \pi_i : C\rangle \qquad \rightarrow\ \langle S(\rho)(o + i - 1) : V, S, E, C\rangle$$

3.9 Case Expressions

The execution of a case expression essentially follows that for a let-expression, except for some added control flow, and a decrement or decomposition of the tested expression. For numeric case, the transition rules are

$$\langle V, S, E, \mathbf{case}\ e\ \mathbf{of}\ 0 \rightarrow e_1\ [\!]\ x + 1 \rightarrow e_2 : C\rangle\ \rightarrow$$
$$\langle V, S, E, e : e_1\ [\!]_x\ e_2 : C\rangle$$
$$\langle 0 : V, S, E, e_1\ [\!]_x\ e_2 : C\rangle\ \rightarrow\ \langle V, S, E, e_1 : C\rangle$$
$$\langle v : V, S, E, e_1\ [\!]_x\ e_2 : C\rangle\ \rightarrow\ \langle v - 1 : V, S, E', e_2 : \mathbf{sret}_1^{C_{ES}}\rangle$$
$$E' \equiv E + \{x \mapsto v - 1\}$$

The instruction $e_1\ [\!]_x\ e_2$ is called the branching instruction. The rules for constructor-case,

$$\langle V, S, E, \mathbf{case}\ e\ \mathbf{of}\ A \cdots A : C\rangle\ \rightarrow\ \langle V, S, E, e : A \cdots A : C\rangle$$
$$\langle o_\rho : V, S, E, \{c_i\ x_{i1} \cdots x_{in_i} \rightarrow e_i\} : C\rangle\ \rightarrow$$
$$\langle v_{n_k} : \cdots : v_1 : V, S, E', e_k : \mathbf{sret}_{n_k}^{C_{ES}}\rangle$$
$$c_k = S(o_\rho);\quad v_j \equiv S((o + j)_\rho);\quad E' \equiv E + \{x_{kj} \mapsto v_j\}$$

are essentially the same: the case-alternative instruction $A_1 \cdots A_n$, which we also write as, $\{c_i\ x_{i1} \cdots x_{in_i} \rightarrow e_i\}$ is a generalisation of $e_1\ [\!]_x\ e_2$: the test refers to the store rather than to the stack; decomposition, as in the rule for tuple-projection, has replaced the decrement.

4 A Type System with Space Effects

A type and effect system is a type system that, in addition to a type, associates a collection of observable side effects with each node of a derivation tree. In MML, there are three kinds of effects: stack effects (δ), put effects (p), and store effects (ϕ). In this section, we shall formulate the type system that lets us infer these effects; we start with a few standard definitions and then proceed to the actual rules.

A *type assignment* Γ is a set of *basic typing statements* of the form $x :: \tau$; a *function type assignment* F is a set of *function typing statements* of the form $f :: \sigma$. By $\Gamma, x :: \tau$ we refer to the set $\{x :: \tau\} \cup \{x' :: \tau' \in \Gamma \mid x' \neq x\}$, and similarly for $\mathrm{F}, f :: \sigma$. A *typing judgement* is a 7-tuple of the form

$$\Gamma \models_{\mathrm{F}} e :: \tau \ !\ \delta;\ p;\ \phi$$

where e, τ, δ, p, and ϕ are given by the grammar in section 2.3. A typing judgement is said to be *valid* if it is derivable using the inference rules of the type system.

A *subtyping judgement* is a pair of the form $\tau_1 \rhd \tau_2$, read as τ_1 is a subtype of τ_2, or of the form $\nu \rhd \nu'$ read similarly. A subtyping judgement is valid if it can be derived using the subtype inference rules of the type system.

The set of valid typing judgements for MML is given by the rules in appendix A; we devote the remainder of this section to their principles.

4.1 Typing Numbers

As we have already seen, the size of a natural number is one plus its value, so the best type we can give a number n is Nat_{n+1}. For sizes, the size of the type is always that of the number itself, i.e, $\check{n} :: \mathsf{Size}_n$. Evaluation of any expression — numbers in particular — will push one element on the value stack, so for numbers, $\delta = 1$; there are no side effects in the store, so $p = \emptyset$ and $\phi = 0$:

$$\Gamma \models_{\mathbb{F}} n :: \mathsf{Nat}_{n+1} \,!\, 1;\ \{\};\ 0 \qquad \Gamma \vdash \check{n} :: \mathsf{Size}_n \,!\, 1;\ \{\};\ 0$$

Because the size s of a type Nat_s is an upper bound on the sizes of the elements in the type, we can also give n any type Nat_s such that $n+1 \leq s$. A number $\check{n}$, however, can only have type Size_n, hence:

$$\frac{}{\mathsf{Nat}_{s_1} \rhd \mathsf{Nat}_{s_2}}\, s_1 \leq s_2 \qquad \mathsf{Size}_s \rhd \mathsf{Size}_s$$

4.2 Typing Constructors

From the set of declarations $\mathbf{D}$, one can easily derive a set of constructor type assumptions $\mathbf{C}$. For our declarations we get

$$\begin{aligned}\mathbf{C} = \{&\mathsf{nil} \ :: \forall\, \mathsf{a\, r} \,.\, \mathsf{r} \rightarrow \mathsf{List_1\, a\, r},\\ &\mathsf{cons} :: \forall\, \mathsf{k\, a\, r} \,.\, \mathsf{a} \times \mathsf{List_k\, a\, r} \times \mathsf{r} \rightarrow \mathsf{List_{k+1}\, a\, r}\}\end{aligned}$$

The effects of a constructor application are given by the rule

$$\frac{\begin{array}{l}\delta \equiv \bigsqcup_{i=1}^{n+1}(i-1+\delta_i);\ \ p \equiv \sum_i p_i + \{\rho \mapsto 1+n\};\ \ \phi \equiv \bigsqcup_i \phi_i\\ c \ :: \ \tau_1 \times \cdots \times \tau_n \times \rho \rightarrow \kappa \qquad \tau_i' \rhd \tau_i \ (\forall i)\\ \Gamma \models_{\mathbb{F}} e_i :: \tau_i' \,!\, \delta_i;\ p_i;\ \phi_i\ (\forall i) \quad \Gamma \models_{\mathbb{F}} e :: \rho \,!\, \delta_{n+1};\ p_{n+1};\ \phi_{n+1}\end{array}}{\Gamma \models_{\mathbb{F}} c\, e_1 \cdots e_n\ e \ :: \ \kappa \,!\, \delta;\ p;\ \phi}$$

When e_1 is evaluated, no value has been pushed on the stack, when e_2 is evaluated one value has been pushed, etc., hence the definition of δ.

The put effect p is the cumulative put-effect of all sub-expressions, plus the effect of writing $1+n$ values to ρ (1 for the c, and n for the v_i's — recall the representation of constructors). Let $p_{\mathbf{R}}$ denote the range of the effect p; the sum of two effects is

$$p + p' = \{\, p(\rho) + p'(\rho) \mid \rho \in (p_{\mathbf{R}} \cup p'_{\mathbf{R}})\,\}$$

The store effect ϕ is simply the maximum store effect of any sub-expression.

4.3 Typing Function Calls

The type inference rule for function calls,

$$\frac{\begin{array}{c}\Gamma \models_{\mathbb{F}} f \ :: \ \tau_1 \times \cdots \times \tau_n \overset{\delta_{n+1};\, p_{n+1};\, \phi_{n+1}}{\rightarrow} \tau\\ \Gamma \models_{\mathbb{F}} e_i \ :: \ \tau_i' \,!\, \delta_i;\ p_i;\ \phi_i \quad (\forall i) \qquad \tau_i' \rhd \tau_i \ (\forall i)\end{array}}{\Gamma \models_{\mathbb{F}} f\, e_1 \cdots\ e_n \ :: \ \tau \,!\, 1 + \bigsqcup_{i=1}^{n+1}(i-1+\delta_i);\ \sum_i p_i;\ \bigsqcup_i \phi_i}$$

is straightforward. In contrast to constructors, the effect of a function call is derived from the function's type. Notice that the stack effect accounts for an activation record on the stack (return address plus actual parameters) during the call.

4.4 Typing Letreg

Any excesses in space use are effectively trapped at the letreg rule. It checks that the put effect of the region it introduces is less than or equal to the size assigned to the region:

$$\frac{\begin{array}{l}\rho \notin FV(\Gamma, \tau); \qquad p'(\rho) \leq s\\ \quad \Gamma \models_{\mathbb{F}} e :: \mathsf{Size}_s \,!\, \delta; p; \phi \qquad \Gamma, x :: \rho \models_{\mathbb{F}} e' :: \tau \,!\, \delta'; p'; \phi'\end{array}}{\Gamma \models_{\mathbb{F}} \mathbf{letreg}\, x\#e\, \mathbf{in}\, e' :: \tau \,!\, \delta \sqcup (1+\delta'); p + p' \backslash \rho; \phi \sqcup (s+1+\phi')}$$

The put effect of the local region ρ should not be visible outside the letreg-expression, so it is removed from the effect set p'. To account for the allocations, the size of the region's frame (s+1) is added to the store effect of the expression that uses it.

The side condition $\rho \notin FV(\Gamma, \tau)$ prohibits deallocations that introduce dangling pointers, e.g.

$$\mathbf{let}\ a = \mathbf{letreg}\ x\#\check{3}\ \mathbf{in}\ (1,1)_x\ \mathbf{in}\ \pi_1\ a + \pi_2\ a$$

4.5 Typing Case Expressions

Two aspects of the numeric case rule,

$$\frac{\begin{array}{l}\delta \equiv \delta_0 \sqcup \delta_1 \sqcup (1+\delta_2); \qquad p \equiv p_0 + (p_1 \sqcup p_2); \qquad \phi \equiv \bigsqcup_i \phi_i\\ \Gamma \models_{\mathbb{F}} e \ :: \ \mathsf{Nat}_{s+1} \,!\, \delta_0;\ p_0;\ \phi_0\\ \Gamma \models_{\mathbb{F}} e_1 \ :: \ \tau_1 \,!\, \delta_1;\ p_1;\ \phi_1\\ \Gamma, x :: \mathsf{Nat}_s \models_{\mathbb{F}} e_2 \ :: \ \tau_2 \,!\, \delta_2;\ p_2;\ \phi_2 \quad \tau_i \rhd \tau\ (\forall i)\end{array}}{\Gamma \models_{\mathbb{F}} \mathbf{case}\ e\ \mathbf{of}\ 0 \rightarrow e_1 \,[\!]\, x+1 \rightarrow e_2 \ :: \ \tau \,!\, \delta;\ p;\ \phi}$$

are worth notice: 1) The binding of the pattern-variable in the second branch (x) requires one word of stack space. The rule accounts for this by adding one to the stack effect of the expression in the second branch. 2) The potential effects of the two branches have to be merged in some way; we do so by taking their least upper bound:

$$p \sqcup p' = \{\, p(\rho) \sqcup p'(\rho) \mid \rho \in (p_{\mathbf{R}} \cup p'_{\mathbf{R}})\,\}$$

The typing rule for constructor-case is, essentially, a generalisation of that for numeric-case. (See app. A.)

4.6 Typing Function Definitions

It remains to relate the function type assignment F, that occurs in the typing judgements for expressions, to the function definitions of a program. For this purpose, we introduce four new kinds of judgements: $F \vdash d :: \nu$ for basic function definitions, $F \vdash g \mapsto F$ for recursive groups of such,

$\bar{g} \Rightarrow F$ for sequences of such groups, and $P :: \tau \mathbin{!} \delta;\ p;\ \phi$ for programs.

The inference rule for the first kind of judgements,

$$\frac{x_1 :: \tau_1, \ldots, x_n :: \tau_n \ \models_{\mathrm{F}}\ e :: \tau \mathbin{!} \delta;\ p;\ \phi}{\mathrm{F}\ \vdash\ f\, x_1 \cdots x_n = e\ ::\ \tau_1 \times \cdots \times \tau_n \overset{\delta;\ p;\ \phi}{\longrightarrow} \tau}$$

is trivial, but the one that uses it,

$$\frac{\begin{array}{c}\sigma_{i[k]} \equiv \forall \bar{k}_i \bar{t}_i \bar{\rho}_i . \nu_{i[k]}; \quad FV(\forall k . \sigma_{i[k]}) = \emptyset; \quad \nu_i' \rhd \nu_{i[k+1]}\ (\forall i) \\ \mathrm{F}, f_1 :: \sigma_{1[k]}, \ldots, f_n :: \sigma_{n[k]} \vdash f_i\ \bar{x}_i = e_i :: \nu_i'\ (\forall i) \quad \sigma_{i[0]} = \mathbf{U}\ (\forall i)\end{array}}{\mathrm{F} \vdash \{f_i\ \bar{x}_i\ =\ e_i\}\ \ \Rightarrow\ \ \mathrm{F}, f_1 :: \forall k\,.\,\sigma_{1[k]}, \ldots, f_n :: \forall k\,.\,\sigma_{n[k]}}$$

is more involved. To guarantee termination, the rule has the same built in induction principle as the letrec-rule of the sized type system (see [HPS96, Par98] for details).

The second premise in the rule for mutual recursion ($FV(\forall k . \sigma_{i[k]}) = \emptyset$) states that the type schemes introduced by the rule must not contain free variables; this is to preserve the invariant "F contains no free variables" which makes the usual side condition for generalisation — a generalised variable must not occur in the type assignment — trivial.

The third and fourth premises together form the induction step. They generalise the idea that the body of a function $f\ ::\ \forall \mathsf{k}\,.\,\sigma_{[k]}$ should have type $f\ ::\ \sigma_{[k+1]}$ under the assumption $f :: \sigma_{[k]}$.

The last premise of the rule ($\sigma_{i[0]} = \mathbf{U}$) constitutes the base case of the built in induction proof. The predicate $= \mathbf{U}$ (see appendix A) holds for universal types, i.e., types that denote all values.

5 Discussions

5.1 Correctness of the Type System

The following property of MML has been proved: if a program is well-typed with the stack effect δ and the store effect ϕ, then the program will use at most $\delta + \phi$ words of memory (neither counting the words necessary to store the program, nor the machine registers).

The first step of this proof is to introduce a notion of size for machine states, written $|M|$, which is defined as the height of the stack plus the sum of the sizes of all region frames:

$$\begin{array}{rcl} |\langle V, S, E, C\rangle| & = & |V| + |S| \\ |v_1 : \cdots : v_n| & = & n \\ |S| & = & \sum_{\rho \in S_{\mathbf{D}}} |S(\rho)| \\ |\{o_1 \mapsto v_1, \ldots, o_k \mapsto v_k\}^n| & = & n \end{array}$$

Environments (E) and instruction sequences (C) are compile time objects, and are thus not included in the size.

The second step is to introduce a type system for machine states, in which both the stack and the store are assigned types. The syntax for the types in this system is

$$\begin{array}{rcl} \mathcal{X} & ::= & \theta \otimes \Delta \\ \theta & ::= & \mathcal{V} {:} \theta \mid \mathcal{V} \\ \mathcal{V} & ::= & \tau \mid \mathcal{C} \\ \mathcal{C} & ::= & \mathcal{X} \overset{\eta;\ \psi}{\longmapsto} \mathcal{X} \\ \Delta & ::= & \{\rho \mapsto u^n, \ldots, \rho \mapsto u^n\} \\ \eta & \in & \textsc{Nat} \quad \psi \in \textsc{Nat} \end{array}$$

A machine type ($\mathcal{X}$) consists of a stack-type (θ) and a store-type (Δ), where a stack-type is a sequence of value types ($\mathcal{V}$) which in turn are either basic types (τ) or continuation types ($\mathcal{C}$).

A continuation type is of the form $\mathcal{X}_1 \overset{\eta;\ \psi}{\longmapsto} \mathcal{X}_2$, where $\mathcal{X}_1$ is the type of the machine state that the continuation expects, $\mathcal{X}_2$ the type of the state that it returns, and η, ψ the *absolute* stack- and store effects (δ and ϕ are *relative* effects).

A store-type (Δ) is a mapping $\{\rho_1 \mapsto u_1^{n_1}, \ldots, \rho_k \mapsto u_k^{n_k}\}$ from region-variables (ρ_i) to pairs u^n, where n denotes the size of the region frame and u the allocations in it.

The third step of the proof is to define the notion of a well-typed machine state using five new typing-judgements: for stacks ($S \vdash V :: \theta$), for stores ($S :: \Delta$), for environments ($S \vdash E :: \Gamma$), for instruction sequences ($\Gamma\ \vdash\ C :: \theta \otimes \Delta \overset{\eta;\ \psi}{\longmapsto} \mathcal{X}$), and for machine states ($M\ ::\ \mathcal{X} \mathbin{!} \eta;\ \psi$). We omit the inference rules for the first four judgements, and just look at the rule for a well-typed machine state:

$$\frac{\begin{array}{c}\eta' \leq \eta \qquad \psi' \leq \psi \\ S \vdash V :: \theta_0 \quad S :: \Delta_0 \quad S \vdash E :: \Gamma \quad \Gamma \vdash C :: \theta_0 \otimes \Delta_0 \overset{\eta';\ \psi'}{\longmapsto} \mathcal{X}\end{array}}{\langle V, S, E, C\rangle :: \mathcal{X} \mathbin{!} \eta;\ \psi}$$

A typing $M\ ::\ \mathcal{X}\ !\ \eta;\ \psi$ tells us that the machine state M, when executed, will reduce to a state of type χ (c.f., $e\ ::\ \mathsf{Nat}_\omega$, which tells us that e will reduce to a natural number.)

The fourth step of the proof is to show that the size of a well typed machine state is less than or equal to the sum of its stack and store effects, i.e, if $M :: \mathcal{X} \mathbin{!} \eta;\ \psi$ is valid, then $|M| \leq \eta + \psi$.

The fifth step is to prove subject reduction, i.e, if $M :: \mathcal{X} \mathbin{!} \eta;\ \psi$ is valid and $M \rightarrow M'$, then $M' :: \mathcal{X} \mathbin{!} \eta;\ \psi$.

Together with a notion of initial machine states, the last two statements let us infer the desired result: let M_0 be the initial machine state for the expression e; if $P :: \tau \mathbin{!} \delta;\ p;\ \phi$ is valid and $M_0 \rightarrow \cdots \rightarrow M_n$, then $\forall i\,.\,|M_i| \leq \delta + \phi$.

A thorough presentation of the proof is beyond the scope of this paper; a correctness proof for an extended version of the type system will appear in the first authors forthcoming PhD-thesis.

5.2 Termination

The statement of the last section tells us that a well typed program runs in constant space, but does not mention termination. However, MML's type system being a strengthening of the sized type system (which does guarantee termination) we have strong reasons to expect that the desired termination proof can be established. This approach includes the development of a denotational model for MML, and a proof that our abstract machine correctly implements this model. Another approach would be to re-develop the theory of sized types in an operational setting.

5.3 Implementation of the Type System

The type system described in this paper has not been implemented, but its predecessor, the sized-type system, has been given an efficient implementation [Par98]; there is no evident reason why our extended type system could not be implemented using the same techniques.

The potential complication of the extension would be subtyping constraints harder to solve than before. In the sized type system the constraints reduce to Presburger-Arithmetic formulae, which we solve using the omega test [Pug92]. We expect this to be possible also in the presence of stack- store- and put-effects.

5.4 Implementation of the Run-Time System

The run-time system of MML has not been implemented either, but as the abstract machine has been designed with conventional compiler implementation techniques in mind, we expect an implementation to be routine.

Let us look at the intended run-time representation of machine states $\langle V, S, E, C\rangle$:

The value stack (V) is implemented using the CPU's stack-pointer as usual, and the store (S) using a separate region stack pointer.

The environment (E) of a machine state mirrors values stored on the run-time stack, but has no run-time representation in addition to this.

The instruction sequence (C) is represented by a program counter that points into a sequence of CPU-instructions which has been generated by the compiler.

To avoid tagging, a value (v) is represented using one machine word only: a number (n or $\check{n}$) is represented by an unsigned integer; an address (o_ρ) by a pointer that points directly to the tuple- or constructor-representation that the address represents; a region (ρ) by a pointer to a region frame; a constructor tag (c) by an integer constant; a continuation (C_{ES}) by a code-pointer. (There is no run-time representation for environments in continuations, nor for their stores — recall our discussions in sec. 3.5).

To verify that a compiler actually can be implemented along these lines — we might have overlooked something in our design of the abstract machine — is future work.

5.5 Trading Efficiency for Predictability

The semantics of MML is very explicit about evaluation order, the size of representations, the layout of the stack and so on. These are factors which optimising compilers often vary, for example changing evaluation order to improve register utilisation. An MML compiler *must not* make such transformations, because they could invalidate the space bounds which our type system guarantees. We must therefore expect MML programs to run somewhat more slowly than those in other languages.

We do not consider this to be a serious problem. Our intended application domain is one where space bounds are essential, and we consider some loss of performance to be an acceptable price to pay.

5.6 Tail Recursion

A well known technique to make recursive programs use less space is to distinguish *tail calls* from arbitrary function calls, and to exploit the fact that tail-calls can be implemented more efficiently. This technique is of particular importance in the context of embedded programming, so in this section we shall study how it can be added to MML.

The idea is to avoid chains of continuations in which a continuation just passes its parameter on to the next. To avoid such chains is to save stack-space, because continuations are represented using the stack.

The function calls that give rise to such chains can be syntactically identified using a notion of *tail expressions*: the body of a function definition is a tail expression; if a case-expression is a tail-expression, then so are the expressions of its branches. If a function call $(f\ e \cdots e)$ is a tail expression, then it can be replaced by a tail call $(f_n\ e \cdots e)$:

$$\begin{array}{lcl} d & ::= & f\ x \cdots x \;=\; b \\ b & ::= & f_n\ e \cdots e \;\mid\; \textbf{case}\ e\ \textbf{of}\ 0 \to b \,[\!]\, x+1 \to b \;\mid\; e \end{array}$$

Tail calls carry an *environment-size* n that is used to discard the environment in which the tail-call occurs. For example, in the tail recursive version of *fac*,

$$\mathit{fac}\ a\ n = \textbf{case}\ n\ \textbf{of}\ 0 \to a \,[\!]\, x+1 \to \mathit{fac}_3\ (n * a)\ x$$

the environment size is 3, for, at the call, the environment binds a, n, and x.

We make tail-calls explicit for the same reason as we made storage management explicit: we believe that the embedded programmer wants this explicit control.

To give meaning to our new construct, we extend the call-instruction of the abstract machine with an environment-size to become $\mathbf{call}_f^n$, and then introduce two new rules:

$$\begin{array}{l} \langle V, S, E, f_k\ e_1 \cdots e_n : C\rangle \quad\to\quad \langle V, S, E, e_1 : \cdots : e_n : \mathbf{call}_f^k\rangle \\ \langle v_n : \cdots : v_1 : v_1' : \cdots : v_k' : C_{ES'} : V, S, E', \mathbf{call}_f^k\rangle \quad\to \\ \qquad\qquad \langle v_n : \cdots : v_1 : C_{ES'} : V, S, \{x_i \mapsto v_i\}, b_f : \mathbf{ret}_n\rangle \end{array}$$

Here, the number n in the second rule is the arity of the function f, which we, as with its body, take to be globally known.

We must also extend our rule for ordinary function calls with respect to our extended instruction:

$$\langle V, S, E, f\ e_1 \cdots e_n : C\rangle \to \langle C_{ES} : V, S, E, e_1 : \cdots : e_n : \mathbf{call}_f^0\rangle$$

To type programs with tail-calls, we introduce a new judgement, $\Gamma \ \vdash_{\mathrm{F}}^{n} b :: \tau\ !\ \delta;\ p;\ \phi$, that accounts for the number of variables in the environment (n). The inference rules for this judgement (which we omit) assert that the appropriate number of values are removed from the stack at any tail-call.

In the extended system we can give *fac* the type

$$\mathit{fac}\ ::\ \forall \mathsf{k}\,.\, \mathsf{Nat}_\omega \times \mathsf{Nat}_\mathsf{k} \xrightarrow{3,\emptyset,0} \mathsf{Nat}_\omega$$

i.e, a type in which the relative stack effect is three words regardless of the recursion depth.

5.7 Region Resetting

A fundamental problem with the letreg-construct is that it only allows for regions with nested lifetimes, and, thus, is ill-suited for computations in which data-lifetimes overlap, e.g., insertion-sorting using

$$\begin{array}{l} \mathit{sort}\ \mathit{xs}\ l\ r = \textbf{case}\ \mathit{xs}\ \textbf{of} \\ \quad \mathsf{nil} \to \mathsf{nil}\ r \\ \quad \mathsf{cons}\ x\ s \to \textbf{letreg}\ t\#l\ \textbf{in}\ \mathit{insert}\ x\ (\mathit{sort}\ s\ l\ t)\ r \end{array}$$

$$\begin{array}{l} \mathit{insert}\ a\ \mathit{xs}\ r = \textbf{case}\ \mathit{xs}\ \textbf{of} \\ \quad \mathsf{nil} \to \mathsf{nil}\ r \\ \quad \mathsf{cons}\ x\ s \to \mathsf{cons}\ (\mathit{min}\ a\ x)\ (\mathit{insert}\ (\mathit{max}\ a\ x)\ s\ r)\ r \end{array}$$

At each recursive call, the function *sort* creates a temporary region to store the sorted tail of its argument list. These lists start their lives at the call to *insert*, *after* the evaluation of the actual parameters, and end their lives at the returns from this call. But the regions to hold these lists start their lives *before* the evaluation of the actuals. This leads to a complexity of $O(n^2)$-space (where n is the length of the list) rather than $O(n)$-space, which one would expect from a sorting function. (It is worth noting that the program above would not even be typable in MML, because of the restriction to programs with linear space behaviour.)

To get around this problem, we extend our language with a construct

$$\textbf{reset}\ e$$

that, given a reference to a region, returns the same reference, but with the side-effect of resetting this region's region-offset to 0:

$$\langle V, S, E, \textbf{reset}\ e : C\rangle \ \rightarrow\ \langle V, S, E, e : \textbf{reset} : C\rangle$$
$$\langle \rho : V, S, E, \textbf{reset} : C\rangle \ \rightarrow\ \langle V, S\backslash\rho + \{\rho \mapsto \{\}^{|S(\rho)|}\}, E, C\rangle$$

With this extension, we can rewrite *sort* to run in linear space, by using two, alternately reset, regions:

$sort\ xs\ r_1\ r_2 = \textbf{case}\ xs\ \textbf{of}$
$\quad \mathsf{nil} \rightarrow \mathsf{nil}\ r_2$
$\quad \mathsf{cons}\ x\ s \rightarrow insert\ x\ (sort\ s\ r_2\ r_1)\ (reset\ r_2)$

With the addition of this construct, our type system needs a new kind of put effect, called *reset effect*, that takes the form of a triple

$$a \mid b \mid c$$

where a, b, and c denote the number of allocations before, between, and after any resets in the program. The put effects of programs without resetting will be natural numbers as before. For example, the put effect of

$\quad \textbf{let}\ x = (3, 4)_r\ \textbf{in}$
$\quad \textbf{let}\ y = (\pi_2\, x, \pi_1\, x)_{\textbf{reset}\ r}\ \textbf{in}$
$\quad \textbf{let}\ z = (5, \pi_1\, y + \pi_2\, y)_r\ \textbf{in}\ z$

is $r \mathrel{+}= 2 \mid 0 \mid 4$.

The only extension of our type system necessary to handle such effects is a new rule for the **reset** construct, and an extension of the operator $+$ in the type system to apply, also, to reset effects. We omit the details of this extension.

To retain our guarantees about termination, we must prevent resetting of regions that contain live data, e.g, the program

$$\textbf{let}\ x = (3, 44)_r\ \textbf{in let}\ y = (x, 4)_{\textbf{reset}r}\ \textbf{in}\ \pi_1(\pi_1\ y))$$

should be illegal, whereas the same program, but with $\pi_2\ y$ as body, should be allowed.

For this purpose, we introduce a *type based region liveness analysis* in which the types, by the use of a universal type $\mathbf{U}$, indicate any dangling references, e.g., y in the program above would have type $[\,\mathbf{U} \times \mathsf{Nat}_5\,]_\rho$, $\pi_1\ y$ the type $\mathbf{U}$, and $\pi_1(\pi_1\ y)$ would not be typable. The definition of this type system is beyond the scope of this paper.

5.8 References and Assignment

We would like to add references and assignment to MML, to show that we can also guarantee space bounds for imperative programs, but unfortunately this is not straightforward. References are already a 'bad case' for Tofte and Talpin's region type system, which cause programs to use unexpectedly much memory, and of course this is reflected in the space bounds we can obtain.

The problem is that, since all values stored in a reference must have the same type, then they must also reside in the same region. This region must therefore have a lifetime at least as long as that of the reference, and so no value stored in a reference can be deallocated until the reference itself has been. For example, every value stored in a counter must be retained until the counter itself is deallocated. This is certainly not the behaviour the programmer intends.

Interestingly, very similar problems arise with lazy datatypes such as streams, and with constructions for explicit concurrency such as communication channels. The elements of a lazy stream, or the values passed over a channel, must naturally all have the same type. Therefore they must all be allocated in the same region, whose lifetime must then be longer than that of the stream or channel respectively. This prevents us deriving space bounds for networks of processes that run continuously, which is unfortunate since these are common in embedded systems.

Here we see the same problem recurring in different guises, which is not so surprising considering that lazy streams and channels can both be implemented in terms of references. One possible solution is to use region resetting, as in the last section; this is how the problem is handled in Tofte's work. We are also investigating 'cyclic regions', which behave as circular buffers, and an effect system to verify that values are dead by the time they are overwritten. This would spare the programmer the need to reset regions explicitly.

5.9 Higher-order Functions

One of the great strengths of functional languages is their support for higher-order programming, which is just as useful in embedded programs as in any other. But MML as we have defined it is a first-order language. However, we see no great difficulties in extending our type system to handle function values also; type-and-effect systems usually work well for higher-order languages.

The only awkwardness is that the effect of a λ-expression must specify how many words are allocated to hold the representation of the function value. If we represent functions closures containing a code pointer and the values of the free variables, then the typing rule for λ would need to specify exactly which variables are built into the closure representation. This is both awkward and perhaps hard for the programmer to predict.

We are therefore planning to introduce higher-order functions via an explicit *partial application* construct instead: for example $f\langle x, y\rangle_r$ would denote the partial application of top-level function f to arguments x and y, and it is easy to see that the representation occupies three words in region r.

5.10 Real Time

Not only does a region based run-time model let us prove that a program runs in constant space, but it also paves the way for functional programming with hard real-time constraints: it eliminates the problem of garbage collection present in traditional implementations of functional languages.

We expect that we, in a way similar to our use of sized types to guarantee execution in bounded space, will be able to use sized types to guarantee execution in bounded time.

5.11 Type Inference

Once we know that type checking for MML is feasible, an immediate question is whether type inference would be feasible too? While this may be so, there are a number of difficulties to be overcome. The type system possesses subtyping, which most likely would lead to incomprehensible error messages, and polymorphic recursion, which is generally hard; moreover, inferring size formulae is a harder problem than the already difficult problem of checking Presburger-arithmetic formulae for truth.

A half-way house would be to make the programmer declare function *types*, but automatically infer their *effects*, thus relieving the programmer of the need to explicitly express the memory usage. We believe this, at least, would be feasible.

6 Conclusions

We have presented a core programming language that guarantees termination and execution in bounded space, that allows user-defined recursive functions and dynamic data-structures, and that does not rely on garbage collection. The language is a strict, first order, functional language, with non-standard constructs for storage-management.

Termination and execution in bounded space is guaranteed by a type and effect system that combines Tofte and Talpin's region types [TT97] with the sized types of Hughes, Pareto, and Sabry [HPS96].

A correctness proof for the type system has been outlined, but not presented in detail: the correctness of the type system is, for termination by the observation that the type system is a strengthening of the sized type system, and for execution in bounded space by a classical subject-reduction proof using abstract machine semantics.

Acknowledgements

We would like to thank Fritz Henglein who ignited this work and Mads Tofte who encouraged us to take it further. Thanks, also, to Jörgen Gustavsson for helpful discussions on abstract machine semantics, and to the anonymous reviewers for their many remarks and suggestions.

References

[AVW93] Joe Armstrong, Robert Virding, and Mike Williams, *Concurrent programming in erlang*, Prentice Hall, 1993.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, *The synchronous dataflow programming language Lustre*, Proc. IEEE **79** (1991), no. 9, 1305–1320.

[HPS96] John Hughes, Lars Pareto, and Amr Sabry, *Proving the correctness of reactive systems using sized types*, POPL '96, January 1996.

[Lan64] P. J. Landin, *The Mechanical Evaluation of Expressions*, Computer Journal **6** (1964), no. 4, 308–320.

[Par98] Lars Pareto, *Sized types*, Licentiate thesis, Chalmers University of Technology, 1998, (http://www.cs.chalmers.se/∼pareto/lic.ps).

[Pug92] William Pugh, *A practical algorithm for exact array dependence analysis*, Communications of the ACM **35** (1992), no. 8, 102–114.

[TBE+] Mads Tofte, Lars Birkedal, Martin Elsman, Niels Hallenberg, Tommy Højfeld Olesen, Peter Sestoft, and Peter Bertelsen, *Programming with regions in the ML Kit*, Tech. Report DIKU-TR-97/12, University of Copenhagen.

[TT97] Mads Tofte and Jean-Pierre Talpin, *Region-based memory management*, Information and Computation **132** (1997), no. 2, 109–176.

A Type System

$$\Gamma \vDash_{\mathrm{F}} n :: \mathsf{Nat}_{n+1} \,!\, 1;\ \{\};\ 0 \qquad \Gamma \vdash \check{n} :: \mathsf{Size}_n \,!\, 1;\ \{\};\ 0 \qquad \Gamma, x :: \tau \vDash_{\mathrm{F}} x :: \tau \,!\, 1;\ \{\};\ 0$$

$$\frac{c :: \tau_1 \times \cdots \times \tau_n \times \rho \to \kappa \qquad \Gamma \vDash_{\mathrm{F}} e_i :: \tau'_i \,!\, \delta_i;\ p_i;\ \phi_i\ (\forall i) \quad \Gamma \vDash_{\mathrm{F}} e :: \rho \,!\, \delta_{n+1};\ p_{n+1};\ \phi_{n+1} \qquad \tau'_i \rhd \tau_i\ (\forall i)}{\Gamma \vDash_{\mathrm{F}} c\ e_1 \cdots e_n\ e\ ::\ \kappa \,!\, \bigsqcup_{i=1}^{n+1}(i-1+\delta_i);\ \sum_i p_i + \{\rho \mapsto 1+n\};\ \bigsqcup_i \phi_i}$$

$$\frac{\Gamma \vDash_{\mathrm{F}} e_1 :: \mathsf{Nat}_{s_1} \,!\, \delta_1;\ p_1;\ \phi_1 \quad \Gamma \vDash_{\mathrm{F}} e_2 :: \mathsf{Nat}_{s_2+1} \,!\, \delta_2;\ p_2;\ \phi_2}{\Gamma \vDash_{\mathrm{F}} e_1 + e_2\ ::\ \mathsf{Nat}_{s_1+s_2} \,!\, \delta_1 \sqcup (1+\delta_2);\ p_1+p_2;\ \phi_1 \sqcup \phi_2} \qquad \frac{\Gamma \vDash_{\mathrm{F}} e_1 :: \mathsf{Nat}_{s_1} \,!\, \delta_1;\ p_1;\ \phi_1 \quad \Gamma \vDash_{\mathrm{F}} e_2 :: \mathsf{Nat}_{s_2} \,!\, \delta_2;\ p_2;\ \phi_2}{\Gamma \vDash_{\mathrm{F}} e_1 - e_2\ ::\ \mathsf{Nat}_{s_1} \,!\, \delta_1 \sqcup (1+\delta_2);\ p_1+p_2;\ \phi_1 \sqcup \phi_2}$$

$$\frac{\Gamma \vDash_{\mathrm{F}} e_i :: \mathsf{Nat}_{s_i} \,!\, \delta_i;\ p_i;\ \phi_i\ \ (\forall i)}{\Gamma \vDash_{\mathrm{F}} e_1 * e_2\ ::\ \mathsf{Nat}_{\omega} \,!\, \delta_1 \sqcup (1+\delta_2);\ p_1+p_2;\ \phi_1 \sqcup \phi_2} \qquad \frac{\Gamma \vDash_{\mathrm{F}} e\ ::\ \tau \,!\, \delta;\ p;\ \phi \qquad \Gamma, x :: \tau \vDash_{\mathrm{F}} e'\ ::\ \tau' \,!\, \delta';\ p';\ \phi'}{\Gamma \vDash_{\mathrm{F}} \mathbf{let}\ x = e\ \mathbf{in}\ e'\ ::\ \tau' \,!\, \delta \sqcup (1+\delta');\ p+p';\ \phi \sqcup \phi'}$$

$$\frac{\Gamma \vDash_{\mathrm{F}} f\ ::\ \tau_1 \times \cdots \times \tau_n \xrightarrow{\delta_{n+1};\ p_{n+1};\ \phi_{n+1}} \tau \qquad \Gamma \vDash_{\mathrm{F}} e_i\ ::\ \tau'_i \,!\, \delta_i;\ p_i;\ \phi_i \quad (\forall i) \qquad \tau'_i \rhd \tau_i\ (\forall i)}{\Gamma \vDash_{\mathrm{F}} f\ e_1 \cdots\ e_n\ ::\ \tau \,!\, 1 + \bigsqcup_{i=1}^{n+1}(i-1+\delta_i);\ \sum_i p_i;\ \bigsqcup_i \phi_i}$$

$$\frac{\Gamma \vDash_{\mathrm{F}} e_i :: \mathsf{Size}_{s_i} \,!\, \delta_i;\ p_i;\ \phi_i\ (\forall i)}{\Gamma \vDash_{\mathrm{F}} e_1 \check{+} e_2 :: \mathsf{Size}_{s_1+s_2} \,!\, \delta_1 \sqcup (1+\delta_2);\ p_1+p_2;\ \phi_2 \sqcup \phi_2} \qquad \frac{\Gamma \vDash_{\mathrm{F}} e_1 :: \mathsf{Size}_{s_1+s_2} \,!\, \delta_1;\ p_1;\ \phi_1 \quad \Gamma \vDash_{\mathrm{F}} e_2 :: \mathsf{Size}_{s_2} \,!\, \delta_2;\ p_2;\ \phi_2}{\Gamma \vDash_{\mathrm{F}} e_1 \check{-} e_2 :: \mathsf{Size}_{s_1} \,!\, \delta_1 \sqcup (1+\delta_2);\ p_1+p_2;\ \phi_1 \sqcup \phi_2}$$

$$\frac{\Gamma \vDash_{\mathrm{F}} e_1 :: \mathsf{Size}_n \,!\, \delta_1;\ p_1;\ \phi_1 \quad \Gamma \vDash_{\mathrm{F}} e_2 :: \mathsf{Size}_s \,!\, \delta_2;\ p_2;\ \phi_2}{\Gamma \vDash_{\mathrm{F}} e_1 \check{*} e_2 :: \mathsf{Size}_{n*s} \,!\, \delta_1 \sqcup (1+\delta_2);\ p_1+p_2;\ \phi_1 \sqcup \phi_2}$$

$$\frac{\Gamma \vDash_{\mathrm{F}} e :: \mathsf{Size}_s \,!\, \delta; p; \phi \qquad \Gamma, x :: \rho \vDash_{\mathrm{F}} e' :: \tau \,!\, \delta'; p'; \phi'}{\Gamma \vDash_{\mathrm{F}} \mathbf{letreg}\, x \# e\, \mathbf{in}\, e' :: \tau \,!\, \delta \sqcup (1+\delta'); p + p' \backslash \rho; \phi \sqcup (s+1+\phi')}\ \rho \notin FV(\Gamma, \tau);\ \ p'(\rho) \leq s$$

$$\frac{\Gamma \vDash_{\mathrm{F}} e_i :: \tau_i \,!\, \delta_i;\ p_i;\ \phi_i\ (\forall i) \quad \Gamma \vDash_{\mathrm{F}} e :: \rho \,!\, \delta_{n+1};\ p_{n+1};\ \phi_{n+1}}{\Gamma \vDash_{\mathrm{F}} (e_1, \ldots, e_n)_e :: [\tau_1 \times \cdots \times \tau_n]_\rho \,!\, \bigsqcup_{i=1}^{n+1}(i-1+\delta_i);\ \sum_i p_i + \{\rho \mapsto n\};\ \bigsqcup_i \phi_i} \qquad \frac{\Gamma \vDash_{\mathrm{F}} e :: [\tau_1 \times \cdots \times \tau_n]_\rho \,!\, \delta;\ p;\ \phi}{\Gamma \vDash_{\mathrm{F}} \pi_i\ e :: \tau_i \,!\, \delta;\ p;\ \phi}$$

$$\frac{\Gamma \vDash_{\mathrm{F}} e\ ::\ \mathsf{Nat}_{s+1} \,!\, \delta_0;\ p_0;\ \phi_0 \qquad \Gamma \vDash_{\mathrm{F}} e_1\ ::\ \tau_1 \,!\, \delta_1;\ p_1;\ \phi_1 \qquad \Gamma, x :: \mathsf{Nat}_s \vDash_{\mathrm{F}} e_2\ ::\ \tau_2 \,!\, \delta_2;\ p_2;\ \phi_2 \quad \tau_i \rhd \tau\ (\forall i)}{\Gamma \vDash_{\mathrm{F}} \mathbf{case}\ e\ \mathbf{of}\ 0 \to e_1\ [\!]\ x+1 \to e_2\ ::\ \tau \,!\, \delta_0 \sqcup \delta_1 \sqcup (1+\delta_2);\ p_0 + (p_1 \sqcup p_2);\ \bigsqcup_i \phi_i}$$

$$\frac{\kappa = c_1\ \bar{\tau}_1 \mid \cdots \mid c_n\ \bar{\tau}_n \quad \Gamma \vDash_{\mathrm{F}} e :: \kappa \,!\, \delta_0; p_0; \phi_0 \quad \Gamma, x_{i1} :: \tau_{i1}, \cdots, x_{in_i} :: \tau_{in_i} \vDash_{\mathrm{F}} e_i :: \tau_i \,!\, \delta_i; p_i; \phi_i\,(\forall i) \quad \tau_i \rhd \tau\,(\forall i)}{\Gamma \vDash_{\mathrm{F}} \mathbf{case}\ e\ \mathbf{of}\ c_1\,\bar{x}_1 \to e_1\ \cdots\ c_n\,\bar{x}_n \to e_n\ ::\ \tau \,!\, \delta_0 \sqcup \bigsqcup_{i=1}^{n}(n_i+\delta_i);\ p_0 + \bigsqcup_{i=1}^{n} p_i;\ \bigsqcup_i \phi_i \delta; p; \phi}$$

$$\frac{\sigma \preccurlyeq \nu}{c :: \nu}\ c :: \sigma \in \mathbf{C} \qquad \frac{\sigma \preccurlyeq \nu}{\Gamma \vDash_{\mathrm{F}} f :: \nu}\ f :: \sigma \in \mathrm{F} \qquad \forall \bar{k}\bar{t}\bar{\rho}\,.\,\nu\ \ \preccurlyeq\ \ \nu[{}^{\bar{s}}/_{\bar{k}}][{}^{\bar{\tau}}/_{\bar{t}}][{}^{\bar{\rho}'}/_{\bar{\rho}}] \qquad \frac{D \preccurlyeq D'}{D'}\ D \in \mathbf{D} \qquad \forall \bar{k}\bar{t}\bar{\rho}\,.\,\varepsilon\ \ \preccurlyeq\ \ \varepsilon[{}^{\bar{s}}/_{\bar{k}}][{}^{\bar{\tau}}/_{\bar{t}}][{}^{\bar{\rho}'}/_{\bar{\rho}}]$$

$$\frac{\bar{g} \Rrightarrow \mathrm{F} \qquad [\,] \vDash_{\mathrm{F}} e :: \tau \,!\, \delta;\ p;\ \phi}{\bar{g}\ e\ ::\ \tau \,!\, \delta;\ p;\ \phi} \qquad \frac{[\,] \vdash g_1 \Rrightarrow \mathrm{F}_1 \quad \cdots \quad \mathrm{F}_{n-1} \vdash g_n \Rrightarrow \mathrm{F}_n}{g_1 \cdots g_n \Rrightarrow \mathrm{F}_n}$$

$$\frac{\mathrm{F}, f_1 :: \sigma_{1[k]}, \ldots, f_n :: \sigma_{n[k]} \vdash f_i\ \bar{x}_i = e_i :: \nu'_i\ (\forall i) \qquad \sigma_{i[0]} = \mathbf{U}\ (\forall i) \qquad \nu'_i \rhd \nu_{i[k+1]}\ (\forall i)}{\mathrm{F} \vdash \{f_i\ \bar{x}_i\ =\ e_i\}\ \ \Rrightarrow\ \ \mathrm{F}, f_1 :: \forall k\,.\,\sigma_{1[k]}, \ldots, f_n :: \forall k\,.\,\sigma_{n[k]}} \qquad \begin{array}{l} \sigma_{i[k]} \equiv \forall \bar{k}_i \bar{t}_i \bar{\rho}_i . \nu_{i[k]} \\ FV(\forall k . \sigma_{i[k]}) = \emptyset \end{array}$$

$$\frac{x_1 :: \tau_1, \ldots, x_n :: \tau_n \vDash_{\mathrm{F}} e :: \tau \,!\, \delta;\ p;\ \phi}{\mathrm{F}\ \vdash\ f\ x_1 \cdots x_n = e\ ::\ \tau_1 \times \cdots \times \tau_n \xrightarrow{\delta;\ p;\ \phi} \tau} \qquad 0 = 0 \qquad \frac{n = 0}{n * i = 0} \qquad \frac{i = 0}{n * i = 0} \qquad \frac{i_1 = 0 \quad i_2 = 0}{i_1 + i_2 = 0}$$

$$\frac{\nu = \mathbf{U}}{\forall k\,.\,\nu = \mathbf{U}} \qquad \frac{\nu = \mathbf{U}}{\forall t\,.\,\nu = \mathbf{U}} \qquad \frac{\nu = \mathbf{U}}{\forall \rho\,.\,\nu = \mathbf{U}} \qquad \frac{\tau_i = \mathbf{E}}{\tau_1 \times \cdots \times \tau_n \xrightarrow{\delta;\ p;\ \phi} \tau = \mathbf{U}}\ \ (\forall i) \qquad \frac{\tau_i = \mathbf{E}}{[\tau_1 \times \cdots \times \tau_n]_\rho = \mathbf{E}}\ (\forall i)$$

$$\mathsf{Nat}_0 = \mathbf{E} \qquad \frac{s_i = 0\ (\forall i \in A) \qquad \tau_i = \mathbf{E}\ (\forall i \in B)}{T\ \bar{s}\ \bar{\tau}\ \rho = \mathbf{E}} \langle A, B \rangle \in T^{=\mathbf{E}} \qquad \frac{}{\mathsf{Nat}_{s_1} \rhd \mathsf{Nat}_{s_2}} s_1 \leq s_2 \qquad \mathsf{Size}_s \rhd \mathsf{Size}_s$$

$$\frac{s_i \rhd s'_i\ (\forall i) \qquad \tau_i \rhd \tau'_i (\forall i)}{T_{s_1 \cdots s_k}\ \tau_1 \cdots \tau_m\ \rho\ \ \rhd\ \ T_{s'_1 \cdots s'_k}\ \tau'_1 \cdots \tau'_m\ \rho} \qquad \frac{}{s_1 \rhd s_2} s_1 \leq s_2 \qquad \frac{\tau_i \rhd \tau'_i\ (\forall i)}{[\tau_1 \times \cdots \times \tau_n]_\rho\ \ \rhd\ \ [\tau_1 \times \cdots \times \tau_n]_\rho}$$

$$\frac{\tau'_i \rhd \tau_i (\forall i) \qquad \tau \rhd \tau'}{\tau_1 \times \cdots \times \tau_n \xrightarrow{\delta;\ p;\ \phi} \tau \rhd \tau'_1 \times \cdots \times \tau'_n \xrightarrow{\delta';\ p';\ \phi'} \tau'} \quad \delta \leq \delta';\ \ p \leq p';\ \ \phi \leq \phi' \qquad \frac{}{p \leq p'}\ p_{\mathbf{D}} \subseteq p'_{\mathbf{D}};\ \forall \rho \in p_{\mathbf{D}}\,.\,p(\rho) \leq p'(\rho)$$