CS 3366 Programming Languages

Prof. R. Muller

The Programming Language Earth

1 Introduction

Earth is a a small pedagogical programming language used in the *CS3366 Programming Languages* course at Boston College. Earth is an extension of Venus. As in Venus, computation in Earth is carried out in the tradition of symbolic systems by simplification rules that drive expressions toward their values (if they have one). Earth adds block-structured recursive functions as well as simple data structures represented with (non-recursive) tuples and variants.

This document discusses some of the issues that arise when these features are present in a programming language. Recursive functions increase both the expressiveness of the language and the complexity of an implementation. As an example, the following Earth-program implements Euclid's greatest common divisor algorithm:

```
let mod(m : int, n : int) : int = if m < n then m else mod(m - n, n) in
let gcd(m : int, n : int) : int =
  let r : int = mod(m, n)
  in
  if r = 0 then n else gcd(n, r)
in
gcd(20, 15)

5
Earth>
```

But the introduction of recursion has a fairly profound effect on the sematics: unlike Venus, there are well-typed Earth-programs that aren't related to any value. A simple example is:

```
F> let f(x : int) : int = f(x) in f(1)
```

This is to say that Earth is strictly more powerful than Venus — it has the power of a Turing machine.

1.1 Functions

Functions are the single most important organizing idiom in programming; they are found, in one form or another, in nearly all high-level programming langauges. Non-recursive functions were introduced in the first high-level programming language, FORTRAN, in the early 1950s.

In order to implement a function on the underlying hardware, storage for the function's local (and temporary) variables must be allocated for the duration of the activation of the function. This block of storage together with linkage information is usually called an *activation record*. Because FORTRAN functions did not allow recursion, a given function could have only one activation, thus its activation record could be allocated by the compiler once and for all in static memory.

1.1.1 Recursion

Recursive functions were introduced in programminig languages in LISP and Algol (circa 1958). LISP was originally conceived as a language for applications in artificial intelligence. It allowed programmers to write recursive functions operating on symbolic expressions — symbols and lists. Algol-60 was an imperative

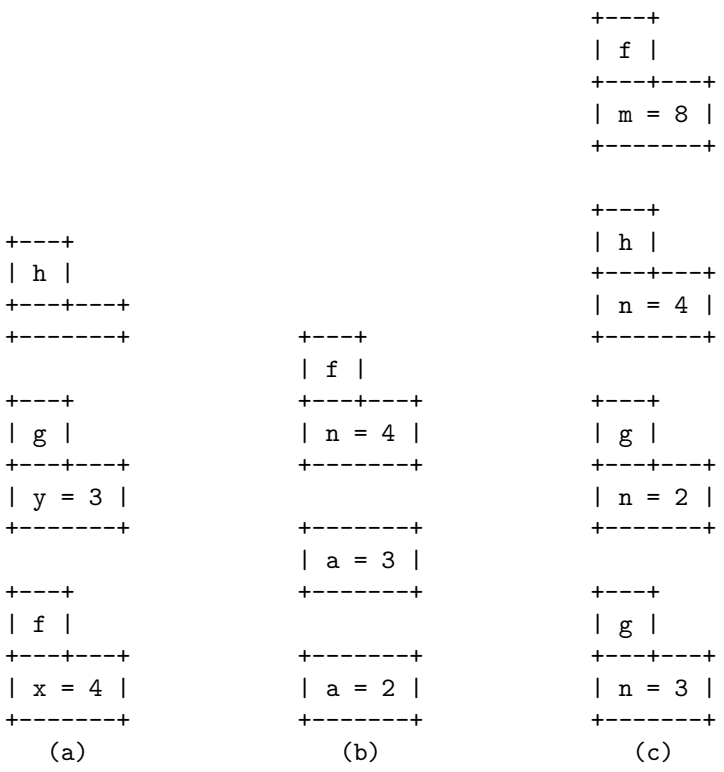


Figure 1: Recursion, the Call Stack and Nested Scopes

programming language in the tradition of FORTRAN. It featured *block-structure*, *lexical* (or *static*) *scoping* and recursive functions.

In and of themselves, recursive functions are relatively easy to implement on stock hardware. The immediate challenge is the management of local storage for functions that might have multiple activations co-existing at the same time. Early on it was understood that the last-in, first-out discipline of a *stack* matched the required lifetimes of activation records for recursive functions. Thus was born the *control* or *call stack* which is central to the implementations of most programming languages.

1.1.2 LISP, the Call Stack and Dynamic Scoping

The early dialects of LISP had a fairly quirky method for resolving the bindings of the free variables that appeared in a function body. Consider the following three function definitions `h`, `g` and `f` in LISP:

```
(defun (h) x)
(defun (g y) (h))
(defun (f x) (g 3))
```

Note that the occurrence of the variable `x` is free with respect to `h`. A call `(f 4)` and freezing the stack after entry to `h` would reveal a call stack of 3 activation records as shown in Figure 1 (a). In looking up the value of `x` during the execution of `h`, LISP scanned through the (dynamic) control stack from top to bottom attempting to find a definition (binding occurrence) of `x`. This stack-based method of resolving free variable references iss called *dynamic scoping*. It's worth noting that a different sequence of function calls ending in a call to `h` might lead to a different variable named `x` being found on the stack.

For many years dynamic scoping was thought to be a reasonable design choice for PL design but most now understand it to be a bug rather than a feature and nearly all modern programming languages use

```
let g(n : int) : int =
  let f(m : int) : int = m + n in
  let h(n : int) : int = f(n * 2)
  in
    if n == 2 then
      h(n * 2)
    else
      g(n - 1) + 1
in
  g(3)
```

Figure 2: Free Variables in Statically-scoped Block-structured PLs

lexical (static) scoping. (Two notable exceptions are *Emacs LISP* and the type-setting language *LaTeX*.) One of the main features of the LISP dialect Scheme, was the adoption of Algol-style lexical scoping.

1.1.3 Block-structured Lexically-Scoped Functions

As we saw in the call-by-name variant of Venus, expressions can have free variable occurrences that are introduced in one binding environment, the *definition* or *static environment*, but which are ultimately evaluated in a different binding environment, the *execution* or *dynamic environment*. In dealing with the call-by-name variant of Venus, we opted to resolve the occurrences of the free variables in the correct (i.e., static) environment by bundling expressions together with the bindings from the definition environment in a *closure* data structure. We'll come back to this shortly, for now, we'll consider stack-based approaches.

Since Earth also features block-structured scope, the free variable problem would remain unchanged in a call-by-name version of Earth. Moreover, the interaction of block-structured scope with nested function definitions causes this to be a problem for call-by-value evaluation in Earth as well. For example in

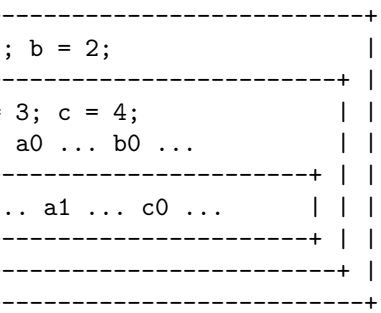
```
let a : int = 2 in
let f(n:int) : int = n + a in
let a : int = 3
in
  f(4)
```

the function `f` is defined in a static environment in which the variable `a` is bound to `2`. However, `f` is applied/called in a dynamic application environment in which `a` is bound to `3`. In order to implement static scoping, the execution of the body of `f` in the dynamic environment must correctly refer to the static binding of `a`.

Correct resolution of static-scoping for stack-based storage is further complicated because free variables can occur in the bodies of recursive functions. Consider the recursive function `g` in Figure 2. The variable `n` occurs free in the body of `f`. The variable is bound in the containing block, the function `g`. When `f` is finally called through the base case of `g`, the dynamic call stack is as depicted in Figure 1 (c). There are multiple activations of `g`, so how do we determine which activation record contains the correct `n`?

Lexical Addresses and Static Chains

The problem of resolving free variables in lexicall-scoped block-structured languages can be solved in a number of different ways. In stack-based implementations of Algol and the Algol-like block-structured languages that followed (e.g., Pascal, PL/I, Ada), a variable occurrence can be represented by a *lexical address* — a pair $(\text{depth}, \text{offset})$, where *depth* is the number of nesting blocks to the corresponding binding occurrence and *offset* is the index of the binding occurrence in the defining block. For example, in



The symbols `a0`, `b0`, `a1` and `c0` have been given indices for reference. These four occurrences of variables `a`, `b` and `c` would be represented by lexical addresses (resp) $(1,0)$, $(0,0)$, $(2,0)$ and $(1,1)$. Note that a lexical address with a depth of 0 means that the variable occurrence is local — it is defined in the same block in which it occurs.

Lexical addresses can be used to access the correct storage for variables allocated in stack-based activation records. In order to facilitate this, activation records include a *static chain* field. A static chain field is a link to the activation record for the nearest containing block. Of course, computing the appropriate link for the static chain field of an activation record was an important part of a the code generated in implementing a function call.

In order to find the storage for a variable with lexical address $(\text{depth}, \text{offset})$, *depth* hops of the static chain links are followed. This leads to the appropriate activation record. Then the variable's storage is found *offset* variables into that local storage.

This scheme "works" in some sense but access to non-local variables tends to be slow, especially if they were deeply nested relative to their binding occurrences. Some optimizing compilers for Algol-like languages addressed this problem by making local copies of non-local variables on entry to the function. The set of local copies of non-local variables is called a *display*. This method reduced the cost of references to non-local variables. But it increased the cost of entering a function (to construct the display) and exiting the function (copying the values of the variables in the display back to their host activation records. It also caused very tricky synchronization problems and greatly complicated error handling if the function threw an exception or otherwise failed to return in the standard way.

One of the most important design decisions in the programming language **C** was to simply rule out nested function definitions. In **C**, function definitions all appear in the same flat namespace. This leads to simpler function calls and faster variable management.

Closures

In some implementations of block-structured languages, the free variables are captured in *closure data structures*. These structures are usually stored in the heap rather than on the stack. (Still other language implementations use *lambda lifting*, which involves adding additional parameters to the nested function and modifying all calls of the function to provide the additional parameters.)

Returning to the example of Figure 2, the nested function `f` might be represented by a closure data structure of the form:

$$\texttt{cf} = \{\texttt{code} = \texttt{fun a} \rightarrow \texttt{ a.arg} + \texttt{a.env.n};\ \texttt{env} = \{\texttt{n} = \texttt{n}\}\}$$

Then, if `cf` is a closure representing the function `f`, an application of the function such as `f(n * 2)` would represented by a closure application:

$$\texttt{cf.code}\ \{\texttt{arg} = \texttt{n} * 2;\ \texttt{env} = \texttt{cf.env}\}$$

The creation of the closure, environment and argument structures requires 3 run-time allocations of memory in the heap (one for each record). And the 4 field projections (i.e., the dots) also have run-time cost (tuple

or record projections correspond to `LOAD` instructions from RAM and accessing RAM is generally slow). Notwithstanding these costs, closure data structures have proven to represent a reasonable solution to the problem of referencing free variables in block-structured languages.

More important, in modern programming languages in which functions can be returned as values, the activation records containing the bindings of free variables might have been popped off the control stack (!) long before the free variables might be used when the function is eventually called. In this case, stack-based approaches are of no help whatsoever and one is left to choose between lamdba lifting or closures. (Most choose the latter.)

1.1.4 Tail-Recursion

The other aspect of stack-based implementations of recursive functions involves the *shape* of the recursive call. The following two functions

```
let fib1(n : int) : int =
  if n < 3 then 1 else fib1(n - 1) + fib1(n - 2)

let fib2(n : int, a : int, b : int) : int =
  if n == 1 then a else fib2(n - 1, b, a + b)
```

compute the same familiar function $\{(1,1),(2,1),(3,2),(4,3),(5,5),\ldots\}$ but the two are obviously quite different. The function `fib1` requires exponential time and logarithmic space. The logarithmic space is due to the accumulation of activation records on the call stack for the recursive calls. The function `fib2` requires linear time and *unit* space. In particular, the call `fib2(n - 1, b, a + b)` is the *last* thing done in the definition of `fib2`. This means that there is no need to retain storage for `fib2`'s variables `n`, `a` and `b`. I.e., activation records do not need to accumulate on the stack. As Guy Steele observed in the 1970s: **a tail-call is a goto that passes parameters.**

When a programming language implementation implements tail-calls in such a way that they do not require needless storage for the parameters and local variables of tail-recursive functions, the implementation is said to have *properly implemented tail-calls*, or sometimes, that they implement the so-called *tail-call optimization*. Another description is that they are *safe for space*.

One of the major differences of the JVM and the .NET VMs is that the latter supports proper tail-calls while the former does not.

1.2 Sum Types

A second important feature of Earth is the presence of *sum* types $\tau_1+\tau_2$. As we'll see for the other types, sum types are associated with expressions (`inl` and `inr`) that *introduce* values of the type as well as a `case` expression that *eliminates* values of sum type. The `case`-form introduces *branching* in the Earth programmer's repetoire. In fact, it is the only way to cause a recursive function to halt in Earth.

Sum types have a long history in PL and there is all too much confusion about how this important type works. (Especially in OO languages.) The essential idea is that sum types provide values that are "either" one type **or** another. For example, in F#, we can define a sum type

```
type Number = Int of int | Float of float

let n : Number = Float 3.14
```

Here, the constructor `Int` can be understood as a function that maps `int`s to `Number`s and the constructor `Float` can be understood as a function that maps `float`s to `Number`s.

Earth does not have a `type` form for *naming* new types, instead there are expression-level forms `inl` and `inr` for introducing values of sum type. The analog of the `Float 3.14` above would be

```
F> let n = inr(3.14, int)
```

```
bool =def= unit+unit

true  =def= inl((), unit)
false =def= inr((), unit)

if e1 then e2 else e3 =def=
  case e1 of inl(x:unit) => e2
           | inr(x:unit) => e3

not e =def=
  case e of inl(x:unit) => false
          | inr(x:unit) => true

e1 || e2 =def=
  case e1 of inl(x:unit) => true
           | inr(x:unit) => e2

e1 && e2 =def=
  case e1 of inl(x:unit) => e2
           | inr(x:unit) => false
```

Figure 3: Sum Types and Booleans

This expression injects `3.14` into the sum type `int+real`. We can then write a `double` function on numbers as follows:

```
let double(m : int+real) : int+real =
  case m of inl(n:int)  => inl(n * 2, real)
          | inr(n:real) => inr(n *. 2.0, int)
```

1.2.1 Booleans, Conditionals and Logical Operators

Sum types are the basis of many constructs that arise in programming. *Enumerate types* (aka *enums*) are a special case of sum types. Figure 3 shows how sum types can be used to define the particular enumerated type *bool* as `unit+unit` with related defintions. It would not be unreasonable for a language implementation to perform the above transformations as part of the elaboration phase of compilation.

1.2.2 Sum Types and the `null` Value

Record types were introduced in programming languages by in 1964 as part of his work on a committee developing a successor language to Algol 60. (The fruit of that committee's work turned out to be Algol 68, a programming language that is generally regarded as a step down from Algol 60.) As Hoare recounts in in the process of developing the truly excellent idea of record types, he also invented the truly terrible idea of the `null` value. The `null` value seemed to be a convenient way to provide an initial value for a variable of record type that wasn't assigned an actual record in it's declaration.

```
type Student = {name:string; id:int};
Student s = {name="Mary"; id=12};
Student t;
```

The `null` value seemed like a convenient value for initializing variables and for providing an implicit base case for recursively defined types. Figure 4 shows snipetts of C and Java with recursively define nodes for a singly-linked list of integers. Because `null` has every type, it can be used as a base case or a termination case for the list.

The utility of `null` in denoting the absence of a value doesn't end there. The `null` value also seemed like a natural value to return as a function return value when the function has no answer to return for a given input. Using a modern-day example, the `get` operation in Java's **Interface Map<K,V>** returns `null` when the map has no answer for the given key:

```
V get(Object key)
```

```
  struct node {
  int info;
  struct node *next;
};
```

C code

```
class Node {
    int info;
    Node next;
    ...
}
```

Java code

Figure 4: The use of null as a base case for a recursively defined data structure

```
Returns the value to which the specified key is mapped, or null if
this map contains no mapping for the key.
```

As he recounts, Hoare was able to persuade the Algol committee to introduce `null` as a convenient jack-of-all-trades value of every type. Both record types and `null` were subsequently adopted in many of the programming languages developed in the decades that followed. They were both featured in **Pascal** and **Ada**. The programming language **C** in particular featured records (as `struct`s) as well as `null`. From here they were drawn into **C++**, **Java**, **Objective-C** etc.

But as Hoare recounts in the above-linked lecture, in hindsight, `null` was an epic mistake. In the 50 years since it was introduced, the `null` value has turned out to be the source of an incalculable number of needless errors in production software. And software that carefully avoids `null`-related errors by explicitly checking for the absence or presence of `null` is needlessly bloated and inefficient.

In modern programming languages, it is understood that the absence or presence of a value of some type is best formulated using sum types. This particular application of sum types is variously called an *option* or *Optional* or*maybe* type.

```
type option<'a> = None    | Some of 'a
type maybe<'a>  = Nothing | Just of 'a
```

A map's `get` operation would then return a value of `option` type as in

```
match (map.get key) with
| None       -> ... code for no mapping for key ...
| Some value -> ... code for key mapping to value ...
```

It's important to note that the *type system* ensures that `value` is a well-defined value that doesn't need to be checked. In contrast, using `null` as a return value:

```
Value value = map.get(key);
if (value == null) then
  ... code for no mapping for key ...
else
  ... code for key mapping to a non-null value of type Value ...
```

even in the `else` clause, the variable `value` is of type `Value` and, since `null` is of type `Value` (`null` is of *every* reference type), all of the places where `value` might flow in the `else`-clause must also explicitly check for the absence or presence of `null`!

It's worth noting that the most recent release of Java, version 8, is attempting to coding style will be hard to uproot.

1.3 Product Types

The last feature of Earth extending Venusis the product type $\tau_1 \times \tau_2$. Intuitively, if `E` is a value of type $\tau_1 \times \tau_2$ then it has both a τ_1 **and** a τ_2 part. So in this sense they are the duals of sum types.

$$
\begin{array}{rcll}
\texttt{i} & \in & \text{integer} & \\
\texttt{r} & \in & \text{real} & \\
\texttt{ValVar} & = & \{\texttt{x}_0, \texttt{x}_1, \ldots\} & \text{Value Variables} \\
\texttt{FunVar} & = & \{\texttt{f}_0, \texttt{f}_1, \ldots\} & \text{Function Variables} \\
\texttt{o} \in \texttt{Operator} & = & \{+, -, *, /, \%, +., -., *., /., =, <, >, \leq, \geq, \texttt{!}\} & \text{Operators} \\
\texttt{u} \in \texttt{Var} & = & \texttt{ValVar} \cup \texttt{FunVar} \cup \texttt{Operator} &
\end{array}
$$

$$
\begin{array}{rcll}
\tau & ::= & \texttt{unit} \mid \texttt{int} \mid \texttt{real} \mid \tau \times \tau \mid \tau + \tau \mid \{a : \tau; \ldots; a : \tau\} \mid \tau \to \tau & \text{Types} \\
\texttt{V} & ::= & () \mid \texttt{i} \mid \texttt{r} \mid (\texttt{V}, \texttt{V}) \mid \texttt{inl}(\texttt{V}, \tau) \mid \texttt{inr}(\texttt{V}, \tau) \mid \{a : \texttt{V}; \ldots; a : \texttt{V}\} & \text{Values} \\
\texttt{E} & ::= & \texttt{V} \mid \texttt{E o E} \mid \texttt{o } E \mid \texttt{i2r(E)} \mid \texttt{r2i(E)} \mid & \text{Terms} \\
 & & (\texttt{E}, \texttt{E}) \mid \texttt{first(E)} \mid \texttt{second(E)} \mid & \\
 & & \texttt{inl}(\texttt{E}, \tau) \mid \texttt{inr}(\texttt{E}, \tau) \mid \texttt{case E of inl}(\texttt{x} : \tau) \Rightarrow \texttt{E} \mid \texttt{inr}(\texttt{x} : \tau) \Rightarrow \texttt{E} & \\
 & & \{a : \texttt{E}; \ldots; a : \texttt{E}\} \mid \texttt{E}.a & \\
 & & \texttt{x} \mid \texttt{f}(\texttt{E}, \ldots, \texttt{E}) \mid \texttt{let D in E} & \\
\texttt{D} & ::= & \texttt{x} : \tau = \texttt{E} \mid \texttt{f}(\texttt{x} : \tau, \ldots, \texttt{x} : \tau) : \tau \ = \ \texttt{E} & \text{Declarations} \\
V_A & ::= & \texttt{V} \mid \texttt{Closure}(\texttt{f}(\texttt{x}, \ldots, \texttt{x}) = \texttt{E}, A) \mid \bar{o} & \text{Environment Values} \\
\Gamma & ::= & \epsilon \mid \Gamma[\texttt{u} : \tau] & \text{Type Environments} \\
\Gamma_0 & = & \epsilon[+ : \texttt{int} \times \texttt{int} \to \texttt{int}] \cdots [\texttt{!} : \texttt{bool} \to \texttt{bool}] & \text{Static Basis} \\
A & ::= & \epsilon \mid A[\texttt{u} \mapsto V_A] & \text{Environments} \\
A_0 & = & \epsilon[+ \mapsto \bar{+}] \cdots [\texttt{!} \mapsto \bar{!}] & \text{Dynamic Basis}
\end{array}
$$

Figure 5: Abstract Syntax of Earth.

Values of product type are usually layed out in memory sequentially. For example, Python's tuples and **C** structs, are represented sequentially.

2 Specification

Abstract Syntax

The basic objects required for the definition of Earth-programs are contained in figure 5. In examples, we'll feel free to use booleans, conditionals etc with the understanding that they have been replaced in the elaboration process as described in Figure 3. As usual, we'll feel free to reuse non-terminal symbols (e.g., τ, E and D) to denote both the sets of elements defined by the non-terminal or as as metavariables ranging over these sets. For example, in some contexts we will use the symbol τ to denote the set of types and in other contexts we will use the symbol τ to represent a typical element of this set. The intended meaning should be clear from the context.

The notation, $\Gamma(\texttt{x})$, denotes the type τ, if it exists, that type environment Γ assigns to identifier $\texttt{x}$. Types are looked up in a type environment as before:

$$
\begin{array}{rcl}
\epsilon(\texttt{x}) & = & \texttt{unbound} \\
\Gamma[\texttt{x} : \tau](\texttt{x}') & = & \begin{cases} \tau & \text{if } \texttt{x} = \texttt{x}' \\ \Gamma(\texttt{x}') & \text{otherwise.} \end{cases}
\end{array}
$$

2.1 Static Semantics

The well-typedness relation is defined by the axoims and inference rules for deriving two forms of *type judgements*:

$$\Gamma \vdash \mathtt{E} : \tau$$

and

$$\Gamma \vdash \mathtt{D} \Rightarrow \Gamma$$

The axoims and inference rules defining the relation are defined in Figures 6, 8 and **??**. Intuitively, if a type judgement $\Gamma \vdash \mathtt{E} : \tau$ is derivable using the axoims and inference rules then the triple $(\Gamma, \mathtt{E}, \tau)$ is in the well-typedness relation.

2.2 Dynamic Semantics

Earth-programs are evaluated relative to a *value environment* A. Value environments are defined analogously to type environments Γ except they map identifiers to values rather than types. The dynamic semantics of Earth-programs is defined by the relations:

$$A \vdash \mathtt{E} \Downarrow V$$

and

$$A \vdash \mathtt{D} \Rightarrow A$$

The $A \vdash \mathtt{E} \Downarrow V$ relation is set of triples relating environment A, to Earth-program `E` and value `V`. It is specified by the *big step* semantics shown in Figures 9 and 10.

2.3 Type Safety

Of course, the name of the game is to establish the connection (or *coherence*) between the static and dynamic semantics. We want to establish that the static semantics (which can be implemented as a compile-time process) is an accurate predictor of the run-time behavior of Earth-programs. The goal is that for any Γ and A that are related in a reasonable way, that for any `E`, `V` and τ such that $\Gamma \vdash \mathtt{E} : \tau$ and $A \vdash \mathtt{E} \Downarrow \mathtt{V}$ it is the case that $\Gamma \vdash \mathtt{V} : \tau$. That is, executing the code didn't change the type predicted by the compiler.

This claim of type safety, sometimes called *soundness*, can be formalized and proved (by induction) as a theorem. But it is most often proved using an alternative specification of the dynamic semantics called *small-step* or *SOS* semantics.

$$\frac{}{\Gamma \vdash \mathtt{i} : \mathtt{int}}\ (\mathtt{int}) \qquad \frac{}{\Gamma \vdash \mathtt{r} : \mathtt{real}}\ (\mathtt{real}) \qquad \frac{}{\Gamma \vdash () : \mathtt{unit}}\ (\mathtt{unit})$$

$$\frac{\Gamma \vdash \mathtt{E} : \mathtt{int}}{\Gamma \vdash \mathtt{i2r(E)} : \mathtt{real}}\ (\mathtt{i2r}) \qquad \frac{\Gamma \vdash \mathtt{E} : \mathtt{real}}{\Gamma \vdash \mathtt{r2i(E)} : \mathtt{int}}\ (\mathtt{r2i})$$

$$\frac{\Gamma(\mathtt{o}) = \tau_1 \to \tau_2;\ \Gamma \vdash \mathtt{E} : \tau_1}{\Gamma \vdash \mathtt{o\ E} : \tau_2}\ (\mathtt{UnOp}) \qquad \frac{\Gamma(\mathtt{o}) = \tau_1 \times \tau_2 \to \tau;\ \Gamma \vdash \mathtt{E}_1 : \tau_1;\ \Gamma \vdash \mathtt{E}_2 : \tau_2}{\Gamma \vdash \mathtt{E}_1\ \mathtt{o}\ \mathtt{E}_2 : \tau}\ (\mathtt{BinOp})$$

$$\frac{\Gamma \vdash \mathtt{E} : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{first(E)} : \tau_1}\ (\times\text{-}\mathtt{E}_1) \qquad \frac{\Gamma \vdash \mathtt{E} : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{second(E)} : \tau_2}\ (\times\text{-}\mathtt{E}_2) \qquad \frac{\Gamma \vdash \mathtt{E}_1 : \tau_1;\ \Gamma \vdash \mathtt{E}_2 : \tau_2}{\Gamma \vdash (\mathtt{E}_1,\ \mathtt{E}_2) : \tau_1 \times \tau_2}\ (\times\text{-}\mathtt{I})$$

$$\frac{\Gamma \vdash \mathtt{E} : \tau_1}{\Gamma \vdash \mathtt{inl(E}, \tau_2) : \tau_1 + \tau_2}\ (+\text{-}\mathtt{I}_1) \qquad \frac{\Gamma \vdash \mathtt{E} : \tau_2}{\Gamma \vdash \mathtt{inr(E}, \tau_1) : \tau_1 + \tau_2}\ (+\text{-}\mathtt{I}_2)$$

$$\frac{\Gamma \vdash \mathtt{E} : \tau_1 + \tau_2;\ \Gamma[\mathtt{x}_1 : \tau_1] \vdash \mathtt{E}_1 : \tau;\ \Gamma[\mathtt{x}_2 : \tau_2] \vdash \mathtt{E}_2 : \tau}{\Gamma \vdash \mathtt{case\ E\ of\ inl(x}_1 : \tau_1) \Rightarrow \mathtt{E}_1 \mid \mathtt{inr(x}_2 : \tau_2) \Rightarrow \mathtt{E}_2 : \tau}\ (+\text{-}\mathtt{E})$$

$$\frac{\Gamma \vdash \mathtt{E}_i : \tau_i}{\Gamma \vdash \{\mathtt{a}_1 = \mathtt{E}_1; \ldots; \mathtt{a}_n = \mathtt{E}_n\} : \{\mathtt{a}_1 : \tau_1; \ldots; \mathtt{a}_n : \tau_n\}}\ (\{\}\text{-}I) \qquad \frac{\Gamma \vdash \mathtt{E} : \{\ldots; \mathtt{a} : \tau; \ldots\}}{\Gamma \vdash \mathtt{E.a} : \tau}\ (\{\}\text{-}E)$$

$$\frac{\Gamma(\mathtt{x}) = \tau}{\Gamma \vdash \mathtt{x} : \tau}\ (\mathtt{Id}) \qquad \frac{\Gamma(\mathtt{f}) = \tau_1 \times \ldots \times \tau_n \to \tau;\ \Gamma \vdash E_i : \tau_i}{\Gamma \vdash \mathtt{f(E}_1, \ldots, \mathtt{E}_n) : \tau}\ (\text{App})$$

$$\frac{\Gamma \vdash D \Rightarrow \Gamma';\ \Gamma' \vdash E : \tau}{\Gamma \vdash \mathtt{let}\ D\ \mathtt{in}\ E : \tau}\ (\text{Let})$$

Figure 6: $\Gamma \vdash \mathtt{E} : \tau$.

$$\frac{\Gamma \vdash \mathtt{E} : \tau}{\Gamma \vdash \mathtt{x} : \tau = \mathtt{E} \Rightarrow \Gamma[\mathtt{x} : \tau]}\ (\text{ValDec})$$

$$\frac{\Gamma[\mathtt{x}_1 : \tau_1] \ldots [\mathtt{x}_n : \tau_n][\mathtt{f} : \tau_1 \times \ldots \times \tau_n \to \tau] \vdash \mathtt{E} : \tau}{\Gamma \vdash \mathtt{f(x}_1 : \tau_1, \ldots, \mathtt{x}_n : \tau_n) : \tau\ =\ \mathtt{E} \Rightarrow \Gamma[\mathtt{f} : \tau_1 \times \ldots \times \tau_n \to \tau]}\ (\text{FunDec})$$

Figure 7: $\Gamma \vdash \mathtt{D} \Rightarrow \Gamma$ ala Church.

$$\frac{\Gamma \vdash \mathtt{E} : \tau}{\Gamma \vdash \mathtt{x} = \mathtt{E} \Rightarrow \Gamma[\mathtt{x} : \tau]}\ (\text{ValDec})$$

$$\frac{\Gamma[\mathtt{x}_1 : \tau_1] \ldots [\mathtt{x}_n : \tau_n][\mathtt{f} : \tau_1 \times \ldots \times \tau_n \to \tau] \vdash \mathtt{E} : \tau}{\Gamma \vdash \mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) \ = \ \mathtt{E} \Rightarrow \Gamma[\mathtt{f} : \tau_1 \times \ldots \times \tau_n \to \tau]}\ (\text{FunDec})$$

Figure 8: $\Gamma \vdash \mathtt{D} \Rightarrow \Gamma$ ala Curry.

$$\frac{}{A \vdash \mathtt{i} \Downarrow \mathtt{i}}\,(\mathtt{int}) \qquad \frac{}{A \vdash \mathtt{r} \Downarrow \mathtt{r}}\,(\mathtt{real}) \qquad \frac{}{A \vdash () \Downarrow ()}\,(\mathtt{unit})$$

$$\frac{A \vdash \mathtt{E} \Downarrow \mathtt{i}}{A \vdash \mathtt{i2r(E)} \Downarrow \mathtt{r}}\,(\mathtt{i2r}) \qquad \frac{A \vdash \mathtt{E} \Downarrow \mathtt{r}}{A \vdash \mathtt{r2i(E)} \Downarrow \mathtt{i}}\,(\mathtt{r2i})$$

$$\frac{A \vdash \mathtt{E} \Downarrow \mathtt{V};\ \mathrm{Apply}_1(\mathtt{o}, \mathtt{V}) = \mathtt{V}'}{A \vdash \mathtt{o\ E} \Downarrow \mathtt{V}'}\,(\mathtt{UnOp}) \qquad \frac{A \vdash \mathtt{E}_1 \Downarrow \mathtt{V}_1;\ A \vdash \mathtt{E}_2 \Downarrow \mathtt{V}_2;\ \mathrm{Apply}_2(\mathtt{o}, \mathtt{V}_1, \mathtt{V}_2) = \mathtt{V}}{A \vdash \mathtt{E}_1\ \mathtt{o}\ \mathtt{E}_2 \Downarrow \mathtt{V}}\,(\mathtt{BinOp})$$

$$\frac{A \vdash \mathtt{E} \Downarrow (\mathtt{V}_1,\ \mathtt{V}_2)}{A \vdash \mathtt{first(E)} \Downarrow \mathtt{V}_1}\,(\times\text{-}\mathtt{E}_1) \qquad \frac{A \vdash \mathtt{E} \Downarrow (\mathtt{V}_1,\ \mathtt{V}_2)}{A \vdash \mathtt{second(E)} \Downarrow \mathtt{V}_2}\,(\times\text{-}\mathtt{E}_2) \qquad \frac{A \vdash \mathtt{E}_1 \Downarrow \mathtt{V}_1;\ A \vdash \mathtt{E}_2 \Downarrow \mathtt{V}_2}{A \vdash (\mathtt{E}_1,\ \mathtt{E}_2) \Downarrow (\mathtt{V}_1,\ \mathtt{V}_2)}\,(\times\text{-}\mathtt{I})$$

$$\frac{A \vdash \mathtt{E} \Downarrow \mathtt{V}}{A \vdash \mathtt{inl}(\mathtt{E}, \tau) \Downarrow \mathtt{inl}(\mathtt{V}, \tau)}\,(+\text{-}\mathtt{I}_1) \qquad \frac{A \vdash \mathtt{E} \Downarrow \mathtt{V}}{A \vdash \mathtt{inr}(\mathtt{E}, \tau) \Downarrow \mathtt{inr}(\mathtt{V}, \tau)}\,(+\text{-}\mathtt{I}_2)$$

$$\frac{A \vdash \mathtt{E} \Downarrow \mathtt{inl}(\mathtt{V}_1, \tau);\ A[\mathtt{x}_1 \mapsto \mathtt{V}_1] \vdash \mathtt{E}_1 \Downarrow \mathtt{V}}{A \vdash \mathtt{case\ E\ of\ inl}(\mathtt{x}_1 : \tau_1) \Rightarrow \mathtt{E}_1 \mid \mathtt{inr}(\mathtt{x}_2 : \tau_2) \Rightarrow \mathtt{E}_2 \Downarrow \mathtt{V}}\,(+\text{-}\mathtt{E})$$

$$\frac{A \vdash \mathtt{E} \Downarrow \mathtt{inr}(\mathtt{V}_2, \tau);\ A[\mathtt{x}_2 \mapsto \mathtt{V}_2] \vdash \mathtt{E}_2 \Downarrow \mathtt{V}}{A \vdash \mathtt{case\ E\ of\ inl}(\mathtt{x}_1 : \tau_1) \Rightarrow \mathtt{E}_1 \mid \mathtt{inr}(\mathtt{x}_2 : \tau_2) \Rightarrow \mathtt{E}_2 \Downarrow \mathtt{V}}\,(+\text{-}\mathtt{E})$$

$$\frac{\begin{array}{c} A \vdash \mathtt{E}_i \Downarrow \mathtt{V}_i \\ A(\mathtt{f}) = \mathtt{Closure}(A', \mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) = \mathtt{E}) \\ A'[\mathtt{x}_i \mapsto \mathtt{V}_i][\mathtt{f} \mapsto \mathtt{Closure}(A', \mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) = \mathtt{E})] \vdash \mathtt{E} \Downarrow \mathtt{V} \end{array}}{A \vdash \mathtt{f}(\mathtt{E}_1, \ldots, \mathtt{E}_n) \Downarrow \mathtt{V}}\,(\mathrm{App})$$

$$\frac{A(\mathtt{x}) = \mathtt{V}}{A \vdash \mathtt{x} \Downarrow \mathtt{V}}\,(\mathtt{Id}) \qquad \frac{A \vdash D \Rightarrow A';\ A' \vdash E \Downarrow \mathtt{V}}{A \vdash \mathtt{let}\ D\ \mathtt{in}\ E \Downarrow \mathtt{V}}\,(\mathrm{Let})$$

Figure 9: $A \vdash \mathtt{E} \Downarrow \mathtt{V}$.

$$\frac{A \vdash \mathtt{E} \Downarrow \mathtt{V}}{A \vdash \mathtt{x} : \tau = \mathtt{E} \Rightarrow A[\mathtt{x} \mapsto \mathtt{V}]}\ (\text{ValDec})$$

$$\frac{}{A \vdash \mathtt{f}(\mathtt{x}_1 : \tau_1, \ldots, \mathtt{x}_n : \tau_n) : \tau \ = \ \mathtt{E} \Rightarrow A[\mathtt{f} \mapsto \mathtt{Closure}(A, \mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) = \mathtt{E})]}\ (\text{FunDec})$$

Figure 10: $A \vdash \mathtt{D} \Rightarrow A$.