CS411 Simple Runtime and Compiling

A. Demers

1 May 2001

In these Notes we give a simple runtime representation for IMPX-like languages. This representation is not "efficient" – an optmimizing compiler person would find it beneath contempt – but it is orders of magnitude better than a direct implementation of our structural semantic evaluation rules.

We then give most of the details of a simple "compiler," that is, a function K that maps IMPX terms to programs that, when executed, produce the values of those terms using the runtime representation we have described.

1 The Implementation Language

The target language for our implementation is intended to be sufficiently low-level that there is an "obvious" translation to machine language, but sufficiently abstract that our programs are not intolerably long. The language is very nearly a subset of C. Here is an informal description.

Datatypes: Expressions are either integer valued or pointer valued. (Integers and pointers are datatypes that should fit in a single word of memory or a single machine register.)

A primitive form of record – essentially just a contiguous sequence of memory locations – is supported in the "heap", described below.

Dynamic Storage Management: Storage management is the one "high-level" feature supported in our target language. We provide a "heap" in which we can allocate records of arbitrary size. (The sizes of all heap-allocated objects are known at compile time, though this is not especially significant).

An expression

p ← new(n)

allocates a record of n contiguous memory locations and returns the address of the first one in p.

The language includes a prefix *dereference* operator, "*", and arithmetic on pointer values. Thus,

*p

denotes the memory word whose address is p; and

*p, *(p+1), . . . , *(p+n-1)

are the n words of the record allocated by the invocation of *textrmnew* in the example above. Subscripted expressions are treated as abbreviations for dereferencing:

p[e] ≡ *(p+e)

This is the form we will use most frequently below.

We've said dereferencing manipulates a single word, without specifying whether that word contains an integer or an address. For our purposes this doesn't matter – memory addresses are in fact represented as integers, and in any case it will be clear from context which meaning is intended.

We assume that any record allocated in the heap is retained as long as it might be needed. This can be achieved by garbage collection: reclaiming the storage occupied by a record only after the record is no longer reachable by any sequence of dereference operations applied to program variables. A technique called *conservative* garbage collection deals properly with the ambiguity between integers and pointers discussed in the preceding paragraph.

Control Structure: The language includes simple conditionals and loops:

if (⟨expression⟩) { ⟨statements⟩ } else { ⟨statements⟩ }
while (⟨expression⟩) { ⟨statements⟩ }

There is no function/procedure call mechanism. However, the language supports a computed goto statement of the form

```
    p ← L
    goto  p
      ...
L:
    ⟨more program statements⟩
```

That is, a program label (L in the above example) may be used as a pointer-valued expression; and a goto statement can use a pointer value (p in the above example) to contain the address of its target.

2 Runtime

The compiler generates code that, when executed, manipulates a runtime state so as to compute correct result values. Here we discuss the overall structure of the runtime and how typed IMPX values are represented in it.

2.1 Overview

We have previously given evaluation rules for IMPX. These rules manipulate configurations of the form

$$\langle e, \phi, \sigma \rangle$$

A computation (that is, a derivation according to the evaluation rules) is roughly a tree labeled by such configurations. Program execution corresponds to a traversal of that tree. The runtime state corresponds to a state of such a traversal. Specifically, the program counter in the executing code determines e, a (sub) expression being evaluated. We maintain an *expression evaluation stack*, a singly-linked list headed by a global variable sp, which holds values produced for subgoals (i.e. subtrees of the derivation) that have been evaluated (i.e. traversed) but whose result values have not yet been used. We maintain an *environment stack*, a singly-linked list headed by a global variable ep, which represents the current environment ϕ. Finally, the store σ is represented implicitly by a collection of heap records reachable from the expression and environment stacks.

2.2 Representing Values

The types in a well typed IMPX program yield useful compile time information about the representations of values. Most IMPX values cannot be represented in

a single machine word, so a value of type τ will be represented as a heap record. The record may be linked directly into the evaluation stack or environment stack; or it may be reachable from one of them by dereferencing operations. It is a property of our type system that all values of a given type τ require the same number of machine words to represent them. The length of the representation of τ can easily be defined by induction on the structure of τ, as follows.

Simple types: These are represented in a single machine word.

$$\text{typelen}[\![\text{int}]\!] = 1$$
$$\text{typelen}[\![\text{bool}]\!] = 1$$

Product types: The fields of a product value are laid out consecutively in memory; so

$$\text{typelen}[\![\mathbf{prod}(\ldots,\ x_i : \tau_i,\ \ldots)]\!] \ = \ \textstyle\sum_i \text{typelen}[\![\tau_i]\!]$$

In addition, we will need the offset of each field from the beginning of the value:

$$\text{offset}[\![\mathbf{prod}(\ldots,\ x_i : \tau_i,\ \ldots).x_j]\!] \ = \ \textstyle\sum_{i=1}^{j-1} \text{typelen}[\![\tau_i]\!]$$

Thus, if p points to (contains the address of) a value of product type τ, then

$$\text{p+offset}[\![\tau.x]\!]$$

points to its x component.

Sum types: A sum type is represented by a tag, which we assume fits in a single word, and a value of one of the element types. Thus,

$$\text{typelen}[\![\mathbf{sum}(\ldots,\ x_i : \tau_i,\ \ldots)]\!] \ = \ 1 + \mathbf{max}_i \ \text{typelen}[\![\tau_i]\!]$$

Thus, if p points to a value of sum type τ, then p points to its tag field, and (p+1) points to its current value.

Function types: Recall in one of our previous operational definitions for IMPX we represented a function value in the environment as a pair

$$\langle e, \phi \rangle$$

where e was the unevaluated function body and ϕ was the environment in which that function body should be evaluated. Pairing a function body with an environment was one way to implement static scope rules, and we shall use it here. The above function value will be represented by a pair

$$\langle ip, ep \rangle$$

of words in memory, where ip is the address of the compiled code for e, and ep is the head pointer of an environment stack representing ϕ.

2.3 Stacks

Our runtime includes several stacks, implemented as follows. A stack is represented by a singly linked list of *frames*. (This is a slight abuse of standard terminology, but only slight). A frame is a record consisting of a link field (at offset 0) followed by a value field (at offset 1) whose length is determined by the typelen of the IMPX value being stored. To allocate a frame that will contain a value of length n, we invoke

$$\text{newframe}(n) \quad \equiv \quad \text{new}(n+1)$$

That is, newframe allocates a record big enough to contain the value and link field. If we define

$$\text{LINK} \quad \equiv \quad 0 \qquad \text{and} \qquad \text{VAL} \quad \equiv \quad 1$$

and let p point to a frame containing a value of some **prod** type τ, then for example

p[LINK]

is the (one word) link field of the stack frame, and

p+VAL+offset$[\![\tau.x]\!]$

is the address of the x field of the stored value.

We introduce a couple of stack manipulation primitives.

PUSH(p, s) $\equiv$ { p[LINK] $\leftarrow$ s; s $\leftarrow$ p }

is a macro that pushes the frame pointed to by p onto the stack whose head pointer is s.

POP(s) $\equiv$ { temp $\leftarrow$ s; s $\leftarrow$ s[LINK]; return(temp) }

is a macro that pops a frame from the stack s and returns the newly popped frame. In most, but not all, cases the popped frame is immediately discarded.

Finally, (because it's not clear where else this belongs) we have a primitive

copy(pto, pfrom, n)

which copies n contiguous words from location pfrom to location pto.

2.4 Stores

There is little to say about the store. Abstractly, the store is a collection of typed locations with addresses. This is implemented below by allocating heap records of the appropriate size, and storing pointers to them in the environment and expression stack.

3 The Compiler

The compiler is a function defined on expressions and ordered type environments (described below). Inductively, the result of

$K[\![e]\!]\pi$

should be some code in the target language. The effect of executing this code, given an initial value of ep that is "consistent" with π, should be to push the value of e on the expression stack sp, correctly updating the values in the store, with no net effect on the environment stack ep.

3.1 Ordered Type Assignments

In addition to an expression e, the compiler function takes an argument π that acts as a type assignment – that is, it contains a typing $(x : \tau)$ for each statically accessible identifier x. However, the type assignments used in our typing rules were unordered sets. For the compiler, type assignments are explicitly sequential. An ordered type assignment behaves like a stack with the top at the right. Lookup proceeds by "walking" the stack from top down until the desired identifier is encountered. Let $\emptyset$ denote the empty assignment, and $\oplus$ be the operation of "pushing" a new assignment. We can define lookup naturally by induction on the ordered assignment:

$$\emptyset(x) = \textbf{error}$$
$$(\pi \oplus (x : \tau))(x) = (x : \tau)$$
$$(\pi \oplus (y : \tau))(x) = \pi(x)$$

To generate code for identifier reference, we shall require the *depth* of x in π, which is defined similarly:

$$\text{depth}(\emptyset, x) = \textbf{error}$$
$$\text{depth}((\pi \oplus (x : \tau)), x) = 0$$
$$\text{depth}((\pi \oplus (y : \tau)), x) = 1 + \text{depth}(\pi, x)$$

Once we have presented the compiler definition, you should convince yourself of the following correspondence between the ordered environment stack π and the runtime environment stack ep. Each represents bindings to a sequence of identifiers – π is a compile time binding of types to a sequence of identifiers, and the environment stack is a run time binding of values to a sequence of identifiers. At all points in the generated code the sequences of identifiers represented by π and ep are *identical*. Thus, π can be thought of as a compile time environment.

3.2 Definition of K

At last we are in a position to define the compilation function. The generated code is a big program in the target language. It does no procedure calls other than the memory and stack management primitives discussed above. It uses only four temporary variables – p, q, r and b. The temporary variables satisfy an important property: no temporary variable is ever live across the code generated by a recursive invocation of K. This guarantees that temporary variable uses in separate clauses of the inductive definition of K will never "step on" one another.

So here is (most of) the definition of K:

Constants:

$K[\![n]\!]\pi =$
p $\leftarrow$ newframe(typelen[[int]]);
p[VAL] $\leftarrow$ n;
PUSH(p, sp);

The generated code simply pushes a new stack frame containing the value n.

Sequential Composition:

$K[\![e_1; e_2]\!]\pi =$
$K[\![e_1]\!]\pi$
POP(sp);
$K[\![e_2]\!]\pi$

The generated code pushes the value of e_1, then pops it, then pushes the result value of e_2.

Binary Operators:

$K[\![e_1 + e_2]\!]\pi =$
$K[\![e_1]\!]\pi$
$K[\![e_2]\!]\pi$
p $\leftarrow$ newframe(typelen[[int]]);
p[VAL] $\leftarrow$ sp[VAL] + sp[LINK][VAL];
POP(sp); POP(sp); PUSH(p, sp);

The code evaluates and stacks e_1 and e_2. It then constructs a new integer stack frame, stores the sum result there, pops the (now useless) frames for e_1 and e_2, and pushes the new result frame.

Identifiers:

$K[\![x]\!]\pi =$
q $\leftarrow$ ep;
q $\leftarrow$ q[LINK]; ... (d times) ... q $\leftarrow$ q[LINK];
p $\leftarrow$ newframe(len);
copy(p+VAL, q+VAL, len);
PUSH(p, sp);

where

$$\mathrm{d} = \mathrm{depth}(\pi, x)$$

and the length value "len" is determined by

$$\tau = \pi(x) \quad \text{and} \quad \mathrm{len} = \mathrm{typelen}[\![\tau]\!]$$

which is known at compile time.

Observe that the depth of x in π at compile time is equal to the number of environment stack frames the code must traverse at run time.

Variable Allocation:

$K[\![\mathbf{newvar}(e)]\!]\pi =$
    $K[\![e]\!]\pi$
    q $\leftarrow$ new(len); copy(q, sp+VAL, len);
    p $\leftarrow$ newframe(1); p[VAL] $\leftarrow$ q;
    POP(sp); PUSH(p, sp);

where as above the length value "len" is determined by

$$\pi \vdash e : \tau \quad \text{and} \quad \mathrm{len} = \mathrm{typelen}[\![\tau]\!]$$

which is known at compile time. Here the record allocated for q (the address of which is returned as the value of the **newvar** expression) is conceptually part of the store; it is not a stack frame, and does not have a LINK field. The arguments of the initializing copy invocation reflect this fact: the source address argument is "p+VAL", representing the value field of the stack frame; while the destination address argument is simply "q".

Variable Reference:

$K[\![e\uparrow]\!]\pi =$
    $K[\![e]\!]\pi$
    p $\leftarrow$ newframe(1); copy(p+VAL, sp[VAL], len);
    POP(sp); PUSH(p, sp);

where the length value "len" is determined by

$\pi \vdash e : \mathbf{var}\ \tau \quad$ and $\quad \text{len} = \text{typelen}[\![\tau]\!]$

which as usual is known at compile time.

Again, look carefully at the two parameters to copy. The first (destination) parameter "p+VAL" is the address of the value portion of the stack frame pointed to by p. The second (source) parameter "sp[VAL]" is the *contents* of the value portion of the stack frame pointed to by sp (that is, the frame at the top of the expression stack). This is the address of a record in the store, as would have been allocated and returned by the code for a **newvar** expression.

Assignment: is an easy exercise.

Record Constructors: (Arrgh!)

$K[\![\langle x_1 \sim e_1, \ldots x_n \sim e_n \rangle]\!]\pi =$
    $K[\![e_1]\!]\pi$; ...; $K[\![e_n]\!]\pi$
    p $\leftarrow$ newframe(len);
    copy(p+VAL+off$_n$, sp+VAL, len$_n$); POP(sp);
        ...
    copy(p+VAL+off$_1$, sp+VAL, len$_1$); POP(sp);
    PUSH(p, sp);

where the length and offset values are determined from the fields in the obvious way:

$\pi \vdash e_i : \tau_i$
$\tau = \mathbf{prod}(\ldots, x_i : \tau_i, \ldots)$
$\text{len}_i = \text{typelen}[\![\tau_i]\!]$
$\text{off}_i = \text{offset}[\![\tau, x_i]\!]$

which are all known at compile time.

Note this code computes the values of the fields in the correct left-to-right order, ensuring that side-effects happen in the correct order, though it then copies the field values into the result record in reverse order.

Field Selection:

$K[\![e.x]\!]\pi =$
    $K[\![e]\!]\pi$
    p $\leftarrow$ newframe(len); copy(p+VAL, sp[VAL+off], len);
    POP(sp); PUSH(p, sp);

The values "len" and "off" are determined by

$$\pi \vdash e : \tau' \qquad \tau' \equiv \mathbf{prod}(ldots\ x : \tau,\ \ldots)$$
$$\text{len} = \text{typelen}[\![\tau]\!] \qquad \text{off} = \text{offset}[\![\tau.x]\!]$$

both of which are known at compile time. Arguably this is very inefficient – an entire record value is pushed onto the stack in order to select any field. However, the semantics requires that, even if it is not pushed on the stack, every field must at least be *computed* if that computation might have side effects.

Sum Types: These are left as exercises.

Conditionals:

$K[\![\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3]\!]\pi =$
    $K[\![e_1]\!]\pi$
    b $\leftarrow$ sp[VAL]; POP(sp);
    if (b == TRUE) {
        $K[\![e_2]\!]\pi$
    } else {
        $K[\![e_2]\!]\pi$
    }

This assumes e_1 is a Boolean expression in IMPX (which it must be if the program is well typed), so its evaluation will have the effect of pushing either TRUE or FALSE on the expression stack. Note that "if(b == TRUE)" and "else" are part of the generated code, as are both of $K[\![e_2]\!]\pi$ and $K[\![e_3]\!]\pi$. The test is performed at run time (as it must be), not at compile time.

Loops: These are similar to conditionals, and are left as another exercise.

Nonrecursive Let Binding:

$K[\![\mathbf{let}\ x \sim e_1\ \mathbf{in}\ e_2]\!]\pi =$
    $K[\![e_1]\!]\pi$
    PUSH(POP(sp), ep);
    $K[\![e_2]\!](\pi \oplus (x : \tau_1))$
    POP(ep);

The type τ_1 is determined by

$\pi \vdash e_1 : \tau_1$

That is, it is the type of the bound expression and hence of the bound variable x. This code has no *net* effect on the environment stack; but it does push the value of e_1 on the environment stack temporarily, while evaluating e_2.

Note the use of "PUSH(POP(sp),ep)" to transfer a frame from the expression stack to the environment. This is one of the few places where the result of POP is not immediately discarded.

Functions: These are by far the most complex case. The trick is to have a clear understanding of the *calling conventions*, essentially the contract or division of labor between the caller and callee. Our convention will be for the function body to be entered with its environment already set up, including the step of pushing the argument onto the new environment. The top of the expression stack will contain an (ip,ep) pair representing the invoker's resumption point – a sort of generalized return address. When the callee returns, it will have pushed its result value onto the expression stack, "covering" the resume (ip,ep) frame. It will be the caller's responsibility to remove this frame.

With this introduction, the case for a function invocation looks like:

$K[\![e_1(e_2)]\!]\pi =$

```
        K[[e1]]π
        K[[e2]]π
        q ← POP(sp);  p ← POP(sp);
        r ← newframe(2);
        r[VAL+EP] ← ep;  r[VAL+IP] ← L;
        ep ← p[VAL+EP];  PUSH(q, ep);
        PUSH(r, sp);
        goto q[VAL+IP];
    L:
        ep ← sp[LINK][VAL+EP];
        p ← POP(sp);  POP(sp);  PUSH(p, sp);
```

How does the generated code work? It first evaluates the function and argument. It puts them into temporaries p and q, and pops them from the expression stack. At this point, p points to a "floating" stack frame containing the (ip,ep) pair for the callee; and q points to a frame holding the argument value for the call. The generated code then constructs the resume (ip,ep) pair – the "generalized return address" mentioned above – and pushes it on the expression stack. It then sets the environment to the "ep" component of the function to be invoked, updates it with the argument q, and jumps to the ip component of the callee.

When the callee returns (to location L) the caller's "epilog" code restores the environment and removes the resume (ip,ep) pair from the expression stack.

Note a minor bug in this presentation: we assume every instance of a generated label L is unique – that is, multiple recursive invocations of K on function applications cannot all be allowed to generate the same label L. Thus, to be truly accurate we need a "gensym()" function that generates and returns a unique new label every time it is invoked. This is not conceptually difficult, but it is slightly messy, so I'll just rely on your good will and intuition here.

The code for a function valued expression is, if not conceptually simpler, at least somewhat shorter.

$K[\![\textbf{lambda}\ x : \tau\ \textbf{dot}\ e]\!]\pi =$

```
        p ← newframe(2);
        p[VAL+IP] ← L1;  p[VAL+EP] ← ep;
        PUSH(p, sp);
        goto L2;
L1:
        K[[e]](π ⊕ (x : τ))
        goto sp[LINK][VAL+IP];
L2:
```

As above, L1 and L2 are globally unique labels. The code to evaluate a λ expression returns a function value on the stack – just an (ip,ep) pair. The code for the function body is generated in-line beginning at label L1, with a branch around it. Thus, the ip component of the expression result is L1, and its ep component is just the current environment.

The function body code is fairly straightforward. According to our calling conventions, it expects to be entered with its argument already pushed onto the environment stack, and with a return (ip,ep) pair at the top of the expression stack. It simply pushes the result value onto the expression stack (the net effect of $K[\![e]\!](\pi \oplus (x : \tau))$ and then returns to the "ip" of the return pair, which is now "buried" under the result value on the expression stack. As noted above, it is the caller's responsibility to remove the return pair from the expression stack.