LETTER

Local Modules in Imperative Languages

Keehang KWON†, Member and Daeseong KANG††, Nonmember

SUMMARY We propose a notion of local modules for imperative languages. To be specific, we introduce a new implication statement of the form $D \supset G$ where $D$ is a module (i.e., a set of procedure declarations) and $G$ is a statement. This statement tells the machine to add $D$ temporarily to the program in the course of executing $G$. Thus, $D$ acts as a local module and will be discarded after executing $G$. It therefore provides efficient program management. In addition, we describe a new constructive module language to improve code reuse. Finally, we describe a scheme which improves the heap management in traditional languages. We illustrate our idea via C, its extension [4] and Java – is an important issue. Yet this has proven a difficult task, relying on ad-hoc techniques such as various cache/page replacement algorithms.

An analysis shows that this difficulty comes from the fact that the machine has no idea as to which modules are to be used in the near future. Therefore module management can be made easier by making the program specify which modules are to be used. Toward this end, inspired by [5]–[7], we propose a new implication statement $D \supset G$, where $D$ is a set of procedure declarations and $G$ is a statement. This has the following execution semantics: add $D$ temporarily to the current program in the course of executing $G$. In other words, the machine loads $D$ to the current program, executes $G$, and then unloads $D$ from the current program. This implication statement is closely related to the `let-expression in functional languages and the implication goals in logic languages [6], [7].

Thus $D$ acts as local procedures to $G$ in that $D$ is hidden from the rest. Our approach calls for a new run-time stack called `program stack`, as it requires run-time loading and unloading. It follows that local procedures in our language is (stack) dynamic scoped in the sense that the meaning of a procedure is always determined by its most recent declaration. On the other hand, most modern languages allow local procedures within nested procedures. These approaches are based on static scoping in granting access and requires no run-time loading and unloading. The main advantages of our approach is the following:

(1) It allows local procedures at the statement level, whereas other languages allow local procedures only at the procedural level. Thus our language provides the programmer more flexibility.

(2) It leads to efficient program/memory management due to loading and unloading. This is not negligible when local procedures are big.

(3) It has a simple, natural syntax and semantics due to dynamic scoping of local procedures. In contrast, it is well-known that other systems have awkward and complicated semantics mainly due to static scoping. Consequently, these systems are very difficult to read, write, implement and reason about.

An analysis shows that this difficulty comes from the fact that the machine has no idea as to which modules are to be used in the near future. Therefore module management can be made easier by making the program specify which modules are to be used. Toward this end, inspired by [5]–[7], we propose a new implication statement $D \supset G$, where $D$ is a set of procedure declarations and $G$ is a statement. This has the following execution semantics: add $D$ temporarily to the current program in the course of executing $G$. In other words, the machine loads $D$ to the current program, executes $G$, and then unloads $D$ from the current program. This implication statement is closely related to the `let-expression in functional languages and the implication goals in logic languages [6], [7].

Thus $D$ acts as local procedures to $G$ in that $D$ is hidden from the rest. Our approach calls for a new run-time stack called `program stack`, as it requires run-time loading and unloading. It follows that local procedures in our language is (stack) dynamic scoped in the sense that the meaning of a procedure is always determined by its most recent declaration. On the other hand, most modern languages allow local procedures within nested procedures. These approaches are based on static scoping in granting access and requires no run-time loading and unloading. The main advantages of our approach is the following:

(1) It allows local procedures at the statement level, whereas other languages allow local procedures only at the procedural level. Thus our language provides the programmer more flexibility.

(2) It leads to efficient program/memory management due to loading and unloading. This is not negligible when local procedures are big.

(3) It has a simple, natural syntax and semantics due to dynamic scoping of local procedures. In contrast, it is well-known that other systems have awkward and complicated semantics mainly due to static scoping. Consequently, these systems are very difficult to read, write, implement and reason about.

On the negative side, it requires a little run-time overhead for loading and unloading. In the sequel, a module is nothing but a set of procedures with a name. Our notion of local procedures extends to a notion of local (occurrences of ) modules in a straightforward way. That is, we propose a new module implication statement $/m \supset G$, where $m$ is a module name and $G$ is a statement. This has the following execution semantics: add a (local occurrence of) module $m$ temporarily to the current program in the course of executing $G$. Note that our modules are stack dynamic in the sense that they are loaded/unloaded in the program in a stack fashion. This leads naturally to the dynamic scoping for procedure names. In contrast, most imperative languages have a module language which is typically based on the notion of static modules with no run-time loading and unloading, leading naturally to static scoping. It is well-known that static scoping causes the naming problem among procedures across independent modules.

Our module system has some advantages over other popular module systems in imperative languages.

(1) It allows the programmer to load and unload other
modules due to the module implication statement. This leads to the dynamic scoping for procedure names. In contrast, this is traditionally impossible in other languages, leading to the static scoping and the naming problem.

(2) Dynamic scoping leads to a simple, natural syntax and semantics, as there is no naming problem.

(3) As we shall see later, it allows mutually recursive modules thanks to dynamic scoping.

In addition, we add a novel module language to improve code reuse. Finally, we propose a variant of the implication statement which considerably simplifies the heap management.

This paper extends a C-like language with the new implication statement. We focus on the minimum core of C. The remainder of this paper is structured as follows. We describe C\textsuperscript{mod}, an extension of C with a new statement in Section 2. In Section 3, we present an example of C\textsuperscript{mod}. In Section 4, we describe a constructive module language for enhancing code reuse. In Section 5, we propose a scheme that improves the heap management. Section 6 concludes the paper.

2. The Core Language

The language is core C with procedure definitions. It is described by G- and D-formulas given by the BNF syntax rules below:

\[
\begin{align*}
G & ::= \text{true} \mid A \mid x = E \mid G \mid G \supset G \\
D & ::= A = G \mid \forall x\ D \mid D \land D
\end{align*}
\]

In the above, \(A\) represents a head of a procedure declaration \(p(x_1, \ldots, x_n)\) where \(x_1, \ldots, x_n\) are parameters. A D-formula is a set of procedure declarations. In the transition system to be considered, a G-formula will function as a statement and a list of D-formulas enhanced with the machine state (a set of variable-value bindings) will constitute a program. Thus, a program is a pair of two disjoint components, i.e., \(\langle D_1 : \ldots : D_n : \text{nil}, \theta \rangle\) where \(D_1 : \ldots : D_n : \text{nil}\) is a stack of D-formulas and \(\theta\) represents the machine state. \(\theta\) is initially empty and will be updated dynamically during execution via the assignment statements.

We will present an interpreter for our language via natural semantics [3]. Note that our interpreter alternates between the execution phase and the backchaining phase. In the execution phase (denoted by \(\text{ex}(P, G, P')\)), it executes a statement \(G\) with respect to \(P\) and produce a new program \(P'\) by reducing \(G\) to simpler forms. The rules (6)-(9) deal with this phase. If \(G\) becomes a procedure call, the machine switches to the backchaining mode. This is encoded in the rule (5). In the backchaining mode (denoted by \(\text{bc}(D, P, A, P')\)), the interpreter tries to find a matching procedure for a procedure call \(A\) inside the module \(D\) by decomposing \(D\) into a smaller unit (via rule (4)-(5)) and reducing \(D\) to its instance (via rule (2)) and then backchaining on the resulting definition (via rule (1)). To be specific, the rule (2) basically deals with argument passing: it eliminates the universal quantifier \(x\) in \(\forall xD\) by picking a value \(t\) for \(x\) so that the resulting instantiation, \([t/x]D\), matches the procedure call \(A\). The notation \(S\) sequd \(R\) denotes the sequential execution of two tasks. To be precise, it denotes the following: execute \(S\) and execute \(R\) sequentially. It is considered a success if both executions succeed. Similarly, the notation \(S\) parand \(R\) denotes the parallel execution of two tasks. To be precise, it denotes the following: execute \(S\) and execute \(R\) in any order. It is considered a success if both executions succeed. The notation \(S \leftarrow R\) denotes reverse implication, i.e., \(R \rightarrow S\).

Definition 1. Let \(G\) be a statement and let \(P\) be the program. Then the notion of executing \((P, G, P')\) and producing a new program \(P'\) is defined as follows:

\[
\begin{align*}
(1) \quad & \text{bc}(A = G_1, P, A, P_1) \leftarrow \text{ex}(P, G_1, P_1). \text{ A matching procedure for } A \text{ is found.} \\
(2) \quad & \text{bc}(\forall xD, P, A, P_1) \leftarrow \text{bc}([t/x]D, P, A, P_1). \text{ A matching procedure for } A \text{ is found.} \\
(3) \quad & \text{bc}(D_1 \land D_2, P, A, P_1) \leftarrow \text{bc}(D_1, P, A, P_1). \text{ A matching procedure for } A \text{ is found.} \\
(4) \quad & \text{bc}(D_1 \land D_2, P, A, P_1) \leftarrow \text{bc}(D_2, P, A, P_1). \text{ A matching procedure for } A \text{ is found.} \\
(5) \quad & \text{ex}(P, A, P_1) \leftarrow (D_1 \in P) \text{ and bc}(D_1, P, A, P_1), \text{ provided that } D_1 \text{ is the first module in the stack, which contains a declaration of } A. \text{ % A is a procedure call} \\
(6) \quad & \text{ex}(P, \text{true}, P). \text{ % True is always a success.} \\
(7) \quad & \text{ex}(\langle S, \theta \rangle, x = E, \langle S, \theta \cup \{(x, E')\} \rangle) \leftarrow \text{eval}(P, E, E'). \text{ % evaluate } E \text{ to get } E'. \text{ Here, } \cup \text{ denotes a set union but } (x, V) \text{ in } \theta \text{ will be replaced by } (x, E'). \\
(8) \quad & \text{ex}(P, G_1; G_2, P_2) \leftarrow \text{ex}(P, G_1, P_1) \text{ sequd } \text{ex}(P_1, G_2, P_2). \text{ % A sequential composition} \\
(9) \quad & \text{ex}(\langle S, \theta \rangle, D \supset G_1, P_1) \leftarrow \text{ex}(\langle D, S, \theta \rangle, G_1, P_1). \text{ % add } D \text{ to the top of the program stack } S. \\
(10) \quad & \text{ex}(\langle S, \theta \rangle, /m \supset G_1, P_1) \leftarrow \text{ex}(\langle D, S, \theta \rangle, G_1, P_1). \text{ % add } D \text{ to the top of the program stack } S.
\end{align*}
\]
If $ex(\mathcal{P}, G; \mathcal{P}_1)$ has no derivation, then the interpreter returns the failure. The rule (9) deals with the new feature.

3. Examples

In our language, a module is simply a set of procedures associated with a name. Below the keyword *module* associates a name to a $D$-formula. The following module $Emp$ has a procedure Age which sets the variable named *age*, whose value represents the employee’s age. Similarly, the module $Bank$ is defined with the procedures Deposit, Withdraw, Balance.

module $Emp$.
Age(emp) =
    switch (emp) {
        case tom: age = 31; break;
        case kim: age = 40; break;
        case sue: age = 22; break;
        default: age = 0; break;
    }
module $Bank$.
Deposit(name,amount) = . . .
Withdraw(name,amount) = . . .
Balance(name) = . . .

Now consider executing the following main statement $G$ from an empty program.

% first task using module EmpAge
( $Emp \Rightarrow$
    (Age(tom); print(age);
     Age(kim); print(age);
     Age(sue); print(age))
);
% second task using module Bank
( $Bank \Rightarrow$
    deposit(tom,$100))

Execution proceeds as follows: Initially the program is empty. Then, the machine loads the module $Emp$ to the program, printing the ages of three employees – Tom, Kim and Sue –, and then unloads the module $Emp$. Then, the machine loads the module $Bank$ to the program, deposits $100 to Tom’s account, and then unloads the module $Bank$. Note that the module $Emp$ is available to the first task only, while $Bank$ to the second task only.

As the second example, let us consider two mutually recursive modules $Ev$ and $Od$. The module $Ev$ has a procedure Even(x) which returns true if x is even. Similarly, the module $Od$ is defined with the procedure Odd(x).

module $Ev$.
Even(x) = if x == 0, true else $Od \Rightarrow$ Odd(x-1);
module $Od$.
Odd(x) = if x == 1, true else $Ev \Rightarrow$ Even(x-1);

Now consider executing even(9) from the module $Ev$. Execution proceeds as follows: Initially the program is empty. Then, the machine loads the module $Emp$ to the program, printing the ages of three employees – Tom, Kim and Sue –, and then unloads the module $Emp$. Then, the machine loads the module $Bank$ to the program, deposits $100 to Tom’s account, and then unloads the module $Bank$. Note that the module $Emp$ is available to the first task only, while $Bank$ to the second task only.

4. A Constructive Module Language

Modern languages typically support code reuse via inheritance. We propose a constructive approach to code reuse as an alternative to inheritance. To begin with, our language provides a special macro function */* which binds a name to a set of method (and constant) declarations. This macro function serves to represent programs in a concise way. For example, given two macro definitions */$p = f(x) = x$ and */$q = g(x) = 0$, the notation */$p \wedge /q$ represents $f(x) = x \land g(x) = 0$. Here $\wedge$ means the accumulation of two modules.

In addition to $\wedge$, our module language provides a
renaming operation of the form \( \text{ren}(b, a)D \) which replaces \( b \) by \( a \) in a module \( D \) and \( \forall xD \) for universal generalization. There are other useful operations such as \text{private} \ f \ D \ (\text{reuse} \ D \ \text{with making} \ f \ \text{private}) \) and \text{share}D \ (\text{reuse} \ D \ \text{via sharing, not copying}) \) but we will not discuss them further here.

Now let us consider macro processing. Macro definitions are typically processed before the execution but in our setting, it is possible to process macros and execute regular programs in an interleaved fashion. We adopt this approach below.

We reconsider the language in Section 2.

\[
G ::= \text{true} | A | x = E | G; G | D \supset G | /n : M \supset G \\
D ::= A = G | /n | \text{ren}(a,b)D | \forall x D | D \land D \\
M ::= /n = D | M \land M
\]

In the above, \( n \) is a name and \( A \) represents a head of a procedure declaration \( p(x_1, \ldots, x_n) \) where \( x_1, \ldots, x_n \) are parameters. A \( D \)-formula is a set of procedure declarations. An \( M \)-formula is called macro definitions and \( \mathcal{M} \) is a list of \( M \)-formulas. In the transition system to be considered, a \( G \)-formula will function as a statement and a list of \( D \)-formulas, a list of \( M \)-formulas and the machine state (a set of variable-value bindings) will constitute a program. Thus, a program is a pair of three disjoint components, i.e., \( \langle D_1 : \ldots : D_n :: \text{nil}, \mathcal{M}, \theta \rangle \) where \( \theta \) represents the machine state, \( \theta \) is initially empty and will be updated dynamically during execution via the assignment statements.

**Definition 2.** Let \( G \) be a statement and let \( \mathcal{P} \) be the program. Then the notion of executing \( \langle \mathcal{P}, G \rangle \) and producing a new program \( \mathcal{P}' \) – \( \text{ex}(\mathcal{P}, G, \mathcal{P}') \) – is defined as follows:

1. \( \text{ex}(\mathcal{P}, G, \mathcal{P}_1) \leftarrow \text{bc}(A = G_1), \mathcal{P}, A, \mathcal{P}_1) \) \% A matching procedure for \( A \) is found.
2. \( \text{bc}(\forall x D, \mathcal{P}, A, \mathcal{P}_1) \leftarrow \text{bc}(\forall x D, \mathcal{P}, A, \mathcal{P}_1) \) \% argument passing
3. \( \text{bc}(D_1 \land D_2, \mathcal{P}, A, \mathcal{P}_1) \leftarrow \text{bc}(D_1 \land D_2, \mathcal{P}, A, \mathcal{P}_1) \) \% look for a matching procedure in \( D_1 \)
4. \( \text{bc}(D_1 \land D_2, \mathcal{P}, A, \mathcal{P}_1) \leftarrow \text{bc}(D_2, \mathcal{P}, A, \mathcal{P}_1) \) \% look for a matching procedure in \( D_2 \)
5. \( \text{bc}(\text{ren}(a,b)D, \mathcal{P}, A, \mathcal{P}_1) \leftarrow \text{bc}(\text{ren}(a,b)D, \mathcal{P}, A, \mathcal{P}_1) \) \% renaming operation
6. \( \text{bc}(\mathcal{P}, \mathcal{P}, A, \mathcal{P}_1) \) if \( \text{bc}(D, \mathcal{P}, A, \mathcal{P}_1) \) and \( (/n = D) \in \mathcal{M} \) \% we assume it chooses the most recent macro definition.

7. \( \text{ex}(\mathcal{P}, A, \mathcal{P}_1) \leftarrow (D_i \in \mathcal{P}) \text{parand} bc(D_i, \mathcal{P}, A, \mathcal{P}_1) \), provided that \( D_i \) is the first module in the stack, which contains a declaration of \( A \) \% \( A \) is a procedure call
8. \( \text{ex}(\mathcal{P}, \text{true}, \mathcal{P}) \). \% True is always a success.
9. \( \text{ex}(\langle \mathcal{S}, \mathcal{M}, \theta \rangle, x = E, \langle \mathcal{S}, \mathcal{M}, \theta \cup \{(x, E')\}\rangle) \leftarrow \text{eval}(\mathcal{P}, E, E') \). \% evaluate \( E \) to get \( E' \). Here, \( \cup \) denotes a set union but \( (x, V) \) in \( \theta \) will be replaced by \( (x, E') \).
10. \( \text{ex}(\mathcal{P}, G_1; G_2, \mathcal{P}_2) \leftarrow \text{ex}(\mathcal{P}, G_1, \mathcal{P}_1) \) seqand \( \text{ex}(\mathcal{P}_1, G_2, \mathcal{P}_2) \). \% a sequential composition
11. \( \text{ex}(\langle \mathcal{S}, \mathcal{M}, \theta \rangle, D \supset G_1, \mathcal{P}_1) \leftarrow \text{ex}(\langle D :: \mathcal{S}, \mathcal{M}, \theta \rangle, G_1, \mathcal{P}_1) \). \% add \( D \) to the top of the program stack \( \mathcal{S} \).
12. \( \text{ex}(\langle \mathcal{S}, \mathcal{M}, \theta \rangle, \mathcal{P} : M \supset G_1, \mathcal{P}_1) \) if \( \text{ex}(\langle \mathcal{P} :: \mathcal{S}, \mathcal{M}, \theta \rangle, G_1, \mathcal{P}_1) \) \% Add new macros to the front of \( \mathcal{M} \). Here :: is a list constructor.

If \( \text{ex}(\mathcal{P}, G, \mathcal{P}_1) \) has no derivation, then the interpreter returns the failure.

5. Improving Heap Management

Our earlier discussions in Section 2 are based on dynamic procedure binding. More interestingly, our notion of implication statements can be applied equally well to static procedure/data binding.

To be specific, allocation and deallocation of (heap) objects – and accessing them through pointers – occur frequently in traditional imperative languages with static procedure/data binding. This includes malloc-free for memory management, new-dispose for objects, new-delete for heap objects (arrays, records, etc). Unfortunately, allocation and deallocation constructs are unstructured. Using allocation and deallocation carelessly leads to serious problems.

Towards an efficient yet robust memory management, we need to impose some restrictions on the use of allocation and deallocation by providing a high-level statement. To be specific, we propose to use the following pointer-implication statement of the form

\[(p = \text{new obj} \supset G)\]

where \( p \) is a pointer and \( \text{obj} \) is a program object or a data object (array, record, etc). This is a variant of the implication statement in Section 1 with the following semantics: It creates an object \( \text{obj} \) with \( p \) being a pointer to \( \text{obj} \), executes the statement \( G \) and then deallocate \( \text{obj} \) and \( p \). To avoid additional complications, we assume that pointers can only be initialized but not manipulated.

The following code is an example where arrayptr1
and int[100] are available in both the statements \( S_1 \) and \( S_2 \), while arrayptr2 and int[1000] are available only in \( S_2 \).

\[
\text{% heap allocation}
\]
\[
\begin{align*}
\text{arrayptr1} &= \text{new int[100]} \triangleright \\
\text{arrayptr2} &= \text{new int[1000]} \triangleright S_1 \quad \triangleright S_2
\end{align*}
\]

Our system requires considerable change to memory management: it needs to maintain three different categories of memory: program/data stack, run-time stack and the heap. Program/data stack is a new component and it – instead of the heap – is used to maintain program/data objects created via the new construct. Thus it replaces most of the works done by the heap. Program/data stack has considerable advantages over the heap: it is more efficient and can simplify several complications caused by the heap including garbage collection, heap fragmentation, and dangling pointers.

6. Conclusion

In this paper, we have proposed a simple extension to imperative languages. This extension introduced an implication statement \( D \triangleright G \) where \( D \) is a module and \( G \) is a statement. This statement makes \( D \) local to \( G \). It therefore maintains only the active modules in the current program context.

7. Acknowledgements

This work was supported by Dong-A University Research Fund.

References

[1] J. Alglave and L. Maranget and M. Tautschnig, “Herding cats: Modelling, simulation, testing and data mining for weak memory”, ACM Transactions on Programming Languages and Systems, vol.36, no.2, pp.1–74, 2014.
[2] G. Boudol and G. Petri, “Relaxed memory models: an operational approach”, In POPL, pp.392–403, ACM, 2009.
[3] G. Kahn, “Natural Semantics”, In the 4th Annual Symposium on Theoretical Aspects of Computer Science, LNCS vol. 247, 1987.
[4] K. Kwon, S. Hur and M. Park, “Improving Robustness via Disjunctive Statements in Imperative Programming”, IEICE Transactions on Information and Systems, vol.E96-D, No.9, pp.2036-2038, September, 2013.
[5] J. Hodas and D. Miller, “Logic Programming in a Fragment of Intuitionistic Linear Logic”, Information and Computation, vol.110, No.2, pp.327-365, 1994.
[6] D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov, “Uniform proofs as a foundation for logic programming”, Annals of Pure and Applied Logic, vol.51, pp.125–157, 1991.
[7] D. Miller, G. Nadathur, Programming with higher-order logic, Cambridge University Press, 2012.