3b – Two Language Kernels (The Kernels)

`while` and `sa-decl`

Mark Armstrong, PhD Candidate, McMaster University

Fall, 2019

1 The imperative "core" – `While`

The basic operation in imperative languages is the *assignment*. To make a Turing-complete language, we additionally need *sequencing* and some *control structures*.

- For Turing completeness, a conditional statement and a potentially infinite loop statement are sufficient.
  - A conditional and a jump (goto) are simpler (and less restrictive), but jumps are hard to reason about.
  - We instead prefer a conditional and a `while` loop.

1.1 Adding reference types

As mentioned, most imperative languages include some ability to work with *references* to memory.

- Even if explicit referencing and dereferencing is rare, we often need to reason about references to implement parts of the language, such as with
  - pass by reference parameter passing, or
  - non-copying assignment.

To introduce reference types to our language, we extend $Expr_0$ to $Expr_0$, adding

- expressions for *referencing* and *dereferencing* variables, and

- variables of *reference type*.

```
⟨expr⟩ ::= ⟨rexpr⟩
⟨rexpr⟩ ::= & var | var
⟨expr⟩ ::= ! var
```

- The `&` operation obtains the reference to a variable.
- The `!` operation dereferences a reference variable, returning the value stored at the reference.
  - If the given `var` is not of reference type, this results in a type error.
    - For the moment, we do not handle type errors.
  - In many languages, the dereferencing operator is `*`.

1.2 A first language based on `while`

So let us define a kernel for imperative languages based on the `while` loop and an `if-then-else`. We'll call this language $While_0$.

```
⟨stmt⟩ ::=
  skip
| var  ⟨expr⟩
| ⟨stmt⟩ ⟨stmt⟩
| if ⟨bexpr⟩ then ⟨stmt⟩ else ⟨stmt⟩
| while ⟨bexpr⟩ do ⟨stmt⟩
```

Note that:

- Sometimes `;` is used to sequence instructions, but this is *abstract syntax*, so we omit it.
  - Similarly, we do not require an `end` marker for the body of `if`'s and `while` loops.
  - We could omit the `then`, `else` and `do` keywords, but choose to keep them for clarity.
- To emphasise that *assignment* is not *equality*, we will write it using the symbol rather than `=`.

1.2.1 The shortcomings of *While*$_0$

While$_0$ is a sufficient language in many ways, but it is missing (at least) two key abstractions.

Subroutines
- Whether they take the form of functions, procedures or a hybrid of the two, subroutines are a highly valuable abstraction.
  - But we can encode them in *While*$_0$ by "inlining".

Scope and lifetime
- *While*$_0$ provides no means to declare variables.
  - Every variable's lifetime is the whole of the runtime.
  - Every variable's scope is the whole of the program.

Since we can encode subroutines as a linguistic abstraction, we do not address the first shortcoming, at least for the moment.

However, to make the kernel language useful, we must address the second shortcoming.

1.3 The *While* language

> Most [imperative] programming languages have, among others, five constructs: assignment, variable declaration, sequence, test and loop. These constructs form the *imperative core* of the language.
>
> — Principles of Programming Languages (Dowek)

We add the "do nothing" command `skip` to this list of constructs to obtain our language *While*.

```
⟨stmt⟩ ::=
  skip
| local var in ⟨stmt⟩
| var  ⟨expr⟩
| ⟨stmt⟩ ⟨stmt⟩
| if ⟨bexpr⟩ then ⟨stmt⟩ else ⟨stmt⟩
| while ⟨bexpr⟩ do ⟨stmt⟩
```

1.4 Embedding *While*

In the kernel language approach,

- there is an implicit assumption that the kernel language is a proper subset of the full programming language.

The syntax of *While* we have given

- is not a proper subset of the syntax of any full programming language
  - (that I am aware of).
  - (To some extent, this is because we have given only abstract syntax).
- *But*, it is close to several,
  - and we should be able to embed *While* programs into a any full imperative programming language.
    - This embedding may not always preserve meaning, though; sometimes the languages don't fully support the abstractions we have in While.
      - Functionally, this means that if we later show translations from practical languages to *While*, embedding and translation may not be inverses of each other.

For interest, let us investigate this embedding with a language we are familiar with:

- Ruby

1.4.1 Embedding *While* into Ruby – Expressions

Starting with expressions,

- all integer and boolean expressions are easily translated into Ruby.
- The refencing operation, `&`, we embed as the method `object_id`.
  - `& x x.object_id`
- The referencing operation, `!`, we embed as the function `ObjectSpace._id2ref`.
  - `! x ObjectSpace._id2ref(x)`

1.4.2 Embedding *While* into Ruby – Statements

Considering each type of statement of *While*:

`skip` • We simply remove all instances of `skip`.

local var in ⟨stmt⟩
: • We embed local variable declaration as the statement `var = nil; s` where `s` is the embedding of the sub-statement.

var = expr
: • We embed assignment as is.

⟨stmt⟩ ⟨stmt⟩
: • We place a semicolon between the embedding of the statements.

if ⟨bexpr⟩ then ⟨stmt⟩ else ⟨stmt⟩
: • We add the keyword `end` after the second statement.

while ⟨bexpr⟩ do ⟨stmt⟩
: • We add the keyword `end` after the statement.

1.4.3 Embedding *While* into Ruby – Example

By our embedding, the *While* program

```
local x in
local y in
x = 5
y = ! & x
while y > 0 do
  y = y - 1
```

is embedded as

```
x = nil ; y = nil ; x = 5 ; y = ObjectSpace._id2ref(x.object_id) ;
while y > 0 do y = y - 1 end
```

2 A declarative model – *SA-Decl*

The second kernel language we consider

- is a proper subset of Oz, and
- contains 8 kinds of statements.

```
⟨stmt⟩ ::=
  skip                                          // Empty statement
| ⟨stmt⟩ ⟨stmt⟩                                 // Sequence
| local ⟨var⟩ in ⟨stmt⟩ end                     // Variable creation
| ⟨var⟩ = ⟨var⟩                                 // Binding
```

```
| ⟨var⟩ = ⟨value⟩                                      // Value creation
| if ⟨var⟩ then ⟨stmt⟩ else ⟨stmt⟩ end                 // Conditional
| case ⟨var⟩ of ⟨pattern⟩ then ⟨s⟩ else ⟨s⟩ end         // Pattern match
| `{` {⟨var⟩}+  `}`                                    // Procedure application
```

3 Where do we go from here?

We will continue working with these kernel languages, beginning by

- providing linguistic abstraction translations for common language features, and
- extending the languages with *types*.

We will also

- define operational semantics for these languages.