3 – Two Language Kernels

`while` and `sa-decl`

Mark Armstrong, PhD Candidate, McMaster University

Fall, 2019

1 Preamble

1.1 Notable references

- Concepts, Techniques, and Models of Computer Programming, Van Roy & Haridi
  - Chapter 2 – Declarative Computation Model
    - Section 2.2 – The single-assignment store
- Principles of Programming Languages, Dowek
  - Chapter 1 – Imperative Core
    - Section 1.3.1 – The Concept of a State
    - Section 1.3.2 – Decomposition of the State
    - Section 1.3.3 – A Visual Representation of a State

1.2 Update history

Sept. 25 • Original version (part a) posted

1.3 Table of contents

2 Language kernels

> In this approach, a practical language is translated into a *kernel language* that consists of a small number of *programmer-significant* elements. The rich set of abstractions and syntax is encoded into the small kernel language. This gives both programmer and student a clear insight into what the language does.
>
> — Concepts, Techniques, and Models of Computer Programming

2.1 A meta-note about exams

Some of these kernel languages will be proper subsets of the languages we use in the course (especially Oz), but some will be theoretical languages.

In midterms and exams, you will generally *not* be expected to have memorised the syntax of such theoretical languages.

In fact (unless it is announced otherwise before the tests), in questions regarding these languages we will provide you (the relevant portions of) the CFG defining the language syntax.

2.2 Justifying the kernel

For a kernel language to be "useful", we must convince ourselves that it includes

- enough abstractions to be practical, but
- not so many that it becomes difficult to reason about.

Almost any general purpose language serves as an example that violates the second condition.

An example of a language which violates the first is the "*Goto*" language. In this language,

- the only type is $\mathbb{N}$ (the natural numbers),
- the only control structure is a conditional *jump*, and
- the only way to modify variables is to increment or decrement them by 1.

```
⟨stmt⟩ ::= [label] ⟨command⟩
       | ⟨stmt⟩ ; ⟨stmt⟩

⟨command⟩ ::= skip
           | var ++
           | var --
           | if-zero var goto label
```

This language is useful for studying computability, but not so much for studying language design.

2.3 Extending the kernel

There are two ways we will extend our kernel languages:

- syntactic sugar, and
- linguistic abstractions.

Syntactic sugar

- is a shortcut notation that can be translated into the kernel language.
- For example, omitting keywords or rearranging expressions.
  - Such as the prefix control structures in Ruby.

Whereas a linguistic abstraction

- is a new abstraction which can be translated into the kernel language.
  - Remember than an abstraction is a tool or device that solves a particular problem.
- For example, adding *functions* to a language with only *procedures*, or `for` loops to a language with only `while` loops.

3 A common set of expressions – $\boldsymbol{Expr}_0$

For simplicity's sake, we will (for the moment) restrict our attention to languages with

- only integer and boolean expressions,
- only integer and *reference* variables,

and we will not assume any type checking.

The *abstract syntax* of these expressions is given by the grammar

```
⟨expr⟩ ::= ⟨bexpr⟩ | ⟨iexpr⟩
⟨bexpr⟩ ::= true | false
         | ⟨expr⟩ == ⟨expr⟩ | ⟨expr⟩ \= ⟨expr⟩
         | ⟨expr⟩ =< ⟨expr⟩ | ⟨expr⟩ < ⟨expr⟩
         | ⟨expr⟩ >= ⟨expr⟩ | ⟨expr⟩ > ⟨expr⟩
⟨iexpr⟩ ::= number | var
         | ⟨iexpr⟩ + ⟨iexpr⟩ | ⟨iexpr⟩ - ⟨iexpr⟩
         | ⟨iexpr⟩ * ⟨iexpr⟩ | ⟨iexpr⟩ div ⟨iexpr⟩ | ⟨iexpr⟩ mod ⟨iexpr⟩
```

3.1 Why subscript zero?

We will later introduce more basic types than just integer and boolean (as well as discussing type checking).

So we call this simple language of expressions $Expr_0$.

4 The memory models

We are going to define two kernel languages; one imperative, and one declarative.

Before we get to the statements of the languages, we need to explain the memory model underlying each.

- The languages are built specifically for their memory models.

We will call these two memory models:

- *Explicit state (with reference types).*
  - The memory model that applies to most imperative languages.
  - So, likely what you are familiar with.
- *Single-assignment store.*
  - A memory model suitable for declarative languages.
  - Particularly useful for its simple support of concurrency.

We begin though by discussing a simpler model, which we call *explicit state without reference types.*

4.1 Assignment vs Binding, Identifier vs Variable

A *binding*

- is an association between a variable and some entity.
  - For instance, a value, or an address/memory reference.
- Generally, we use the term *binding* to mean a *permanent binding*; once created, such a binding never changes.
- We instead use *assignment* to refer to non-permanant bindings.

In the simplest memory models, values are simply assigned to variables; memory is a mapping between variable names and values.

In general, things are more complex, and it is useful to distinguish the terms

- *identifier* to refer to the names of variables as they appear in the program source, and
- *variable* to refer to the abstract notion of a cell of memory.

4.2 States, environments

We will generally denote the contents of memory using the symbols σ (sigma) and (tau), possibly with subscripts (σ_1, $_2$, etc.) or primes (σ', ''', etc.).

We use notation similar to substitution to *update* memory states.

- `σ[x := v]` is read as σ with `x` updated to be `v`.

We will also need to discuss an *environment* for variables and procedures/functions; we will use Σ (Sigma), adorned with subscripts or primes, to denote environments.

4.3 Explicit state without reference types

In the explicit state without reference types model, we can view

- state as a mapping from identifiers to values.

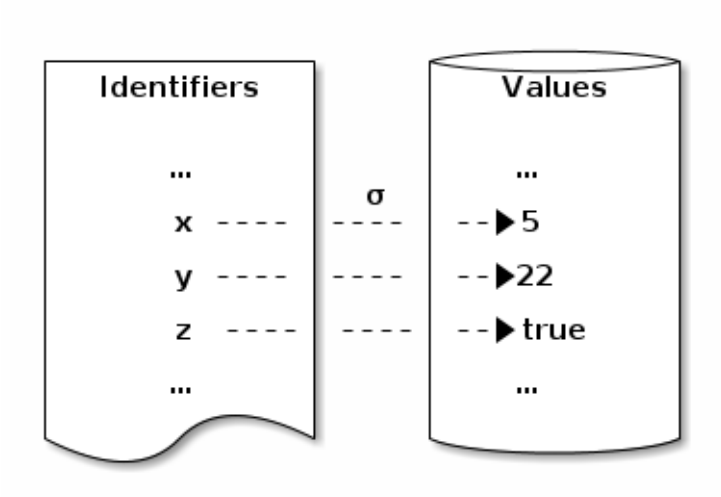
So a state σ is a mapping between identifiers and values.

- `σ : Identifier → Value`

This is a nice, simple mental model.

- But it is *too* simple; most imperative languages include a *reference type* abstraction that cannot be represented by this model.

4.3.1 Visualisation of explicit state without reference types



If σ is this state, the update σ`[x := v]` changes the arrow coming from `x` to instead point to value `v`.

4.4 Explicit state with reference types

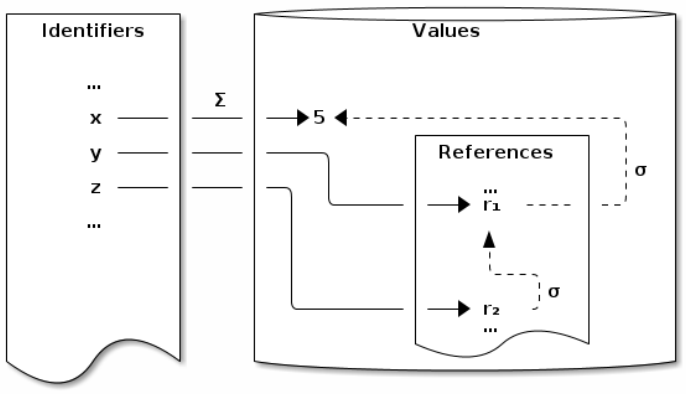
When we add the concept of *references* or *addresses* to our memory model, we must update our notion of state; we now view

- state as a mapping from *references* to values.
  - Rather than from *identifiers* to values.
- We then need an *environment* to map identifiers to references.

The addition of references to our model provides the abstractions

- *constants*, as identifiers mapped to directly to values,
- *reference type variables*, as identifiers mapped to a reference *which maps to a reference*.

4.4.1 Visualisation of explicit state with reference types



In this memory configuration, `x` is a constant, `y` is a (simple) variable, and `z` is a *reference* variable.

Assignment changes the arrows leaving *references*; the arrows leaving identifiers are fixed.

So if this state is σ, σ`[`$\mathtt{r}_1$ `:= v]` would be a new state where $\mathtt{r}_1$ points to `v` instead of `5`.

4.5 Single-assignment store

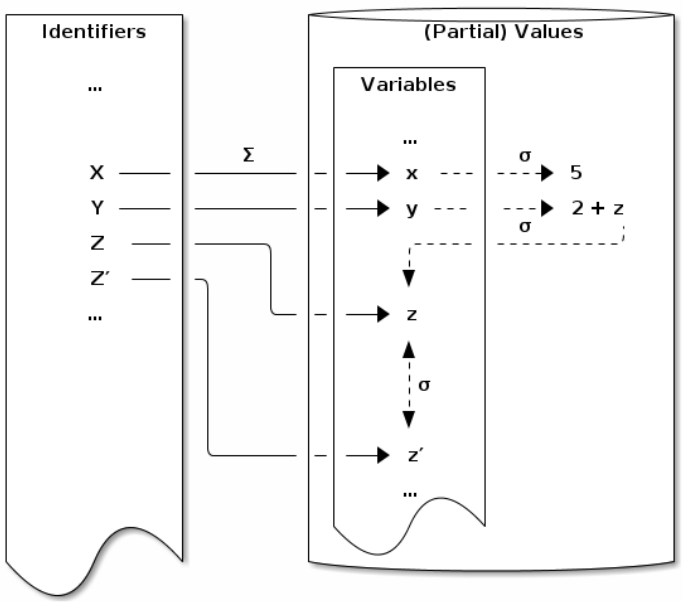
In a single-assignment store, variables can be bound to values *at most once*.

- Once (totally) bound, a variable is *indistinguishable* from the value it is bound to.

We consider a single-assignment store model which allows *partial bindings*.

- A value can involve variables.
  - In which case we call it a *partial* value.
  - If a value is *just* a variable, we call the binding a *variable-variable* binding.
- The variables in a value may be bound later.
  - A variable bound to a partial value is called *partially bound*.
  - As opposed to a *totally bound* variable.

4.5.1 Visualisation of single assignment store



In this memory configuration,

- variable `x` is (totally) bound to the value `5`,
- variable `y` is (partially) bound to the value `2 + z`,
- variable `z` and `z'` are bound together, but neither is bound to a value.

5 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.

5.1 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 `*`.

5.2 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 `=`.

5.2.1 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.

5.3 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⟩
```

5.4 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

5.4.1 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)`

5.4.2 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.

5.4.3 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
```

6 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
```

7 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.