The **while** language

Given test b, expression e over signature Σ:

skip

$x{:=}e$

$S_1;S_2$

if b **then** S_1 **else** S_2 **fi**

while b **do** S_0 **od**

$WP(\Sigma) =$ set of all **while** programs over Σ.

program	Euclid
signature	Naturals_for_Euclidean_Algorithm
sorts	nat,bool
constants	0: $\rightarrow$nat;
	true,false: $\rightarrow$bool
operations	mod: nat $\times$ nat$\rightarrow$nat;
	$\neq$: nat $\times$ nat$\rightarrow$bool
endsig	

```
body
var        x,y,r: nat
begin
           read x;
           r:=x mod y;
           while r ≠ 0 do
                 x:=y; y:=r; r:=x mod y
           od;
           write y
end
```

Problem of Semantics

To model mathematically what happens when any **while** *program performs a computation over* any *data type.*

To give a mathematical definition of how a **while** *program computes a function on any data type.*

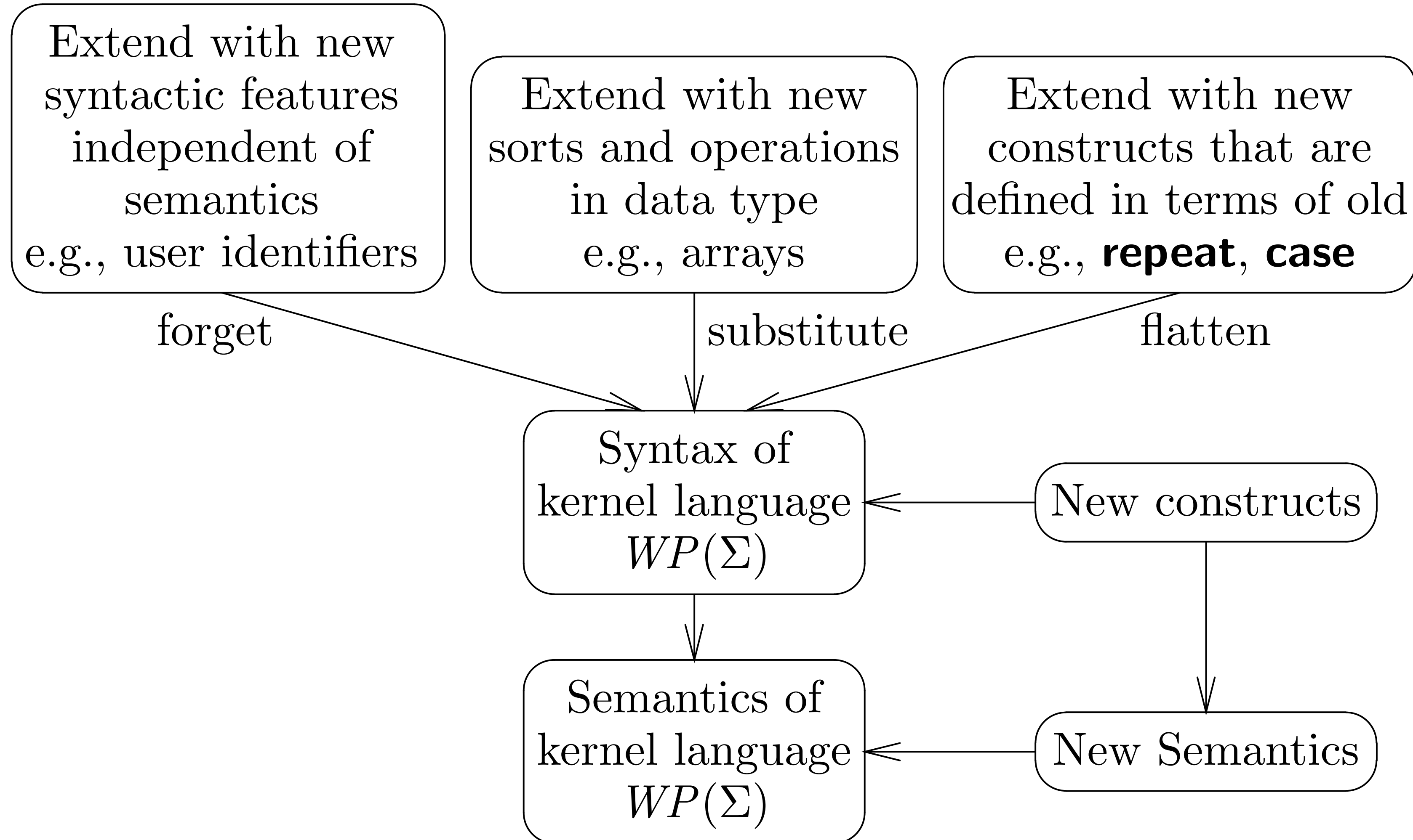
Let Σ be a signature modelling the interface of a data type. Let $WP(\Sigma)$ be the set of all **while** programs over Σ.

We will build a mathematical model

input-output semantics

whose equations allow us to derive the *output* of a program from its *input*.

Extending **while** language



Semantics of a Simple Construct

Assignment 1: **begin var** x,y:nat; x:=y **end**

"Make the value of y the new value of x. The value of y is not changed but the old value of x is lost."

Assignment 2: **begin var** x,y,z:nat; x:=y+z **end**

"Evaluate the sum of the data that are the values of y and z, and make this the new value of x. The values of y and z are not changed but the old value of x is lost."

Assignment 3: **begin var** x,y,z:nat; x:=$(y+z)/2$ **end**

"Evaluate the sum of the data that are the values of y and z. Divide by *2* and, if this number is a natural number, then make this the new value of x. The values of y and z are not changed but the old value of x is lost. If however the division leads to a rational then ..."

Assignment 4: **begin var** x,y,z:real; x:=sqr$[\![(y+z)/2]\!]$ **end**

"Evaluate the sum of the data that are the values of y and z, and divide by 2. Take the square root of this datum, if it exists, and make this the new value of x. The values of y and z are not changed but the old value of x is lost. If the square root does not exist then ..."

input-output **behaviour of a program**

Suppose the program S is over a data-type with signature Σ, i.e., $S \in WP(\Sigma)$. Suppose the data type is implemented by a Σ-algebra A. We will define the concept of a *state* of a computation using data from the algebra A and hence the set

$$State(A)$$

of *all possible* states of all possible computations using data from A.

The *input-output behaviour* of a program S is specified by a function

$$M_A^{io}(S) : State(A) \rightarrow State(A)$$

such that for any state $\sigma \in State(A)$

$$M_A^{io}(S)(\sigma) = \text{the final state of the computation generated by a program } S \text{ from an initial state } \sigma\text{, if such a final state exists.}$$

Termination

Since a **while** loop may execute forever, we do not expect the function $M_A^{io}(S)$ to be defined on all states. That is, we expect $M_A^{io}(S)$ to be a partial function. If there is a final state τ of the computation of S on σ we say the computation is terminating and we write

$$M_A^{io}(S)(\sigma)\downarrow \quad \text{or} \quad M_A^{io}(S)(\sigma)\downarrow \tau$$

otherwise it is non-terminating and we write

$$M_A^{io}(S)(\sigma)\uparrow .$$

To model the behaviour of programs, we will solve the following problem:

> **Problem of Input-Output Semantics** *To give a* precise *mathematical definition of the input-output function* $M_A^{io}(S)$.

signature Σ

sorts $\ldots, s, \ldots, Bool$

constants $\ldots, c : \quad \rightarrow s, \ldots$

$true, false : \rightarrow Bool$

operations $\ldots, f : \quad s(1) \times \cdots \times s(n) \rightarrow s, \ldots$

$\ldots, r : \quad t(1) \times \cdots \times t(m) \rightarrow Bool, \ldots$

$not : \quad Bool \rightarrow Bool$

$and, or : \ Bool \times Bool \rightarrow Bool$

endsig

algebra	A
carriers	$\ldots, A_s, \ldots, \mathbb{B}$
constants	$\ldots, c^A : \quad \rightarrow A_s, \ldots$
	$true^A, false^A : \rightarrow \mathbb{B}$
operations	$\ldots, f^A : \quad A_{s(1)} \times \cdots \times A_{s(n)} \rightarrow A_s, \ldots$
	$\ldots, r^A : \quad A_{t(1)} \times \cdots \times A_{t(m)} \rightarrow \mathbb{B}, \ldots$
	$not^A : \quad \mathbb{B} \rightarrow \mathbb{B}$
	$and^A, or^A : \mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

signature	$Peano$	
sorts	$nat, Bool$	
constants	$zero :$	$\to nat$
	$true, false :$	$\to Bool$
operations	$succ :$	$nat \to nat$
	$add, mult :$	$nat \times nat \to nat$
	$not :$	$Bool \to Bool$
	$and :$	$Bool \times Bool \to Bool$
	$less_than :$	$nat \times nat \to Bool$

signature	$Ordered_field_of_reals$
sorts	$real, Bool$
constants	$zero, one : \ \rightarrow real$
	$true, false : \rightarrow Bool$
operations	$add, minus, mult, div : \ real \times real \rightarrow real$
	$not : \ Bool \rightarrow Bool$
	$and : \ Bool \times Bool \rightarrow Bool$
	$less_than : \ real \times real \rightarrow Bool$

States

The data of A belong to the family

$$\langle A_s \mid s \in S \rangle$$

of carrier sets of A. Each sort of data needs its own store.

We will consider **while** programs that operate over some S-sorted family

$$Var = \langle Var_s \mid s \in S \rangle$$

of variables, where

Var_s is the set of all variables of sort s.

For each sort $s \in S$ an *s-state over* A is a map:

$$\sigma_s : Var_s \to A_s$$

which represents a possible configuration of a store of data of sort s from A. The idea is that

$$\sigma_s(x) = \textit{ value in } A_s \textit{ of variable } x \in Var_s.$$

Let $State_s(A)$ be the set of all s-sorted states over A.

A *state over* A is a family

$$\sigma = \langle \sigma_s \mid s \in S \rangle$$

of s-states over A and represents a possible configuration of a complete store of data from A.

The set of all states over A represents all configurations of this abstract state over A, and is given by

$$State(A) = \langle State_s(A) \mid s \in S \rangle.$$

$\vdots$

sort s	variables $x_0^s, x_1^s, \ldots, x_n^s, \ldots$
state σ_s	values $\sigma_s(x_0^s)$, $\sigma_s(x_1^s)$, $\ldots$, $\sigma_s(x_n^s)$, $\ldots$

$\vdots$

Substitutions in States

Let $\sigma = \langle \sigma_s \mid s \in S \rangle$ be a state over A. Let $s \in S$, $x \in Var_s$ and $a \in A_s$. To change the value of a variable x in the state σ to the new value a we require a substitution operation that transforms σ_s into a new state

$$\sigma_s[a/x]$$

This substitution is defined by:

$$\sigma_s[a/x](y) = \begin{cases} \sigma_s(y) & \text{if } y \neq x; \\ a & \text{otherwise.} \end{cases}$$

So σ_s is unchanged except that the new value of x is a and the old value is lost; in particular, for $y \neq x$,

$$\sigma_s[a/x](y) = \sigma_s(y)$$

Example

Var_{real}	r_1	r_2	r_3	r_4	$\cdots$	r_i	$\cdots$
$\sigma_{real}(r_i)$	0	1.5	π	$\sqrt{2}$	$\cdots$	-4	$\cdots$
$\sigma_{real}[3.14/r_3](r_i)$	0	1.5	3.14	$\sqrt{2}$	$\cdots$	-4	$\cdots$

Expressions

We define the *value* of an expression e on a state σ over A by means of the function

$$V_A : Exp(\Sigma) \to (State(A) \to A)$$

where for $e \in Exp(\Sigma)$ the purpose of the function

$$V_A(e) : State(A) \to A$$

is, for $\sigma \in State(A)$

$$V_A(e)(\sigma) = \text{ the value in } A \text{ of expression } e \text{ on state } \sigma.$$

In detail, V_A is a family

$$\langle V_A^s \mid s \in S \rangle$$

of functions

$$V_A^s : Exp_s(\Sigma) \rightarrow (State_s(A) \rightarrow A_s)$$

that define the value of expressions of sort s on states of sort s.

The family V_A is defined by induction on the structure of terms over Σ simultaneously for each sort $s \in S$ by:

$$
\begin{aligned}
V_A^s(c)(\sigma) &= c^A \\
V_A^s(x)(\sigma) &= \sigma(x) = \sigma_s(x) \\
V_A^s(f(e_1, \ldots, e_n))(\sigma) &= f^A(V_A^{s(1)}(e_1)(\sigma), \ldots, V_A^{s(n)}(e_n)(\sigma))
\end{aligned}
$$

Example Expression Evaluation

If

$$\sigma_{real}(x) = \mathit{1} \quad \text{and} \quad \sigma_{real}(y) = \mathit{3.14}$$

then

$$\begin{aligned} V_A^{real}(add(x,y))(\sigma) &= +(\,V_A^{real}(x)(\sigma),\, V_A^{real}(y)(\sigma)) \\ &= +(\sigma_{real}(x), \sigma_{real}(y)) \\ &= +(\mathit{1}\,, \mathit{3.14}\,) \\ &= \mathit{4.14} \end{aligned}$$

Tests

The semantics of Boolean expressions is the special case V_A^{Bool} of the semantics of expressions. We define the *value* of a Boolean expression b on a state σ over A by means of

$$W_A : BExp(\Sigma) \rightarrow (State(A) \rightarrow \mathbb{B})$$

where for $b \in BExp(\Sigma)$ the purpose of the function

$$W_A(b) : State(A) \rightarrow \mathbb{B}$$

is, for $\sigma \in State(A)$

$$W_A(b)(\sigma) = \text{ value in } \mathbb{B} \text{ of Boolean expression } b \text{ on state } \sigma.$$

We give an inductive definition on the syntactic structure

$$W_A(\mathbf{true})(\sigma) = tt$$

$$W_A(\mathbf{false})(\sigma) = \mathit{ff}$$

$$W_A(\mathbf{r}(e_1, \ldots, e_n))(\sigma) = r^A(V_A(e_1)(\sigma), \ldots, V_A(e_n)(\sigma))$$

$$W_A(\mathbf{not}(b))(\sigma) = \begin{cases} tt & \text{if } W_A(b)(\sigma) = \mathit{ff}; \\ \mathit{ff} & \text{if } W_A(b)(\sigma) = tt. \end{cases}$$

$$W_A(\mathbf{and}(b_1, b_2))(\sigma) = \begin{cases} tt & \text{if } W_A(b_1)(\sigma) = tt \text{ and} \\ & \quad W_A(b_2)(\sigma) = tt; \\ \mathit{ff} & \text{otherwise.} \end{cases}$$

Statements and Commands: First Definition

The input-output semantics for commands is given by the following functions:

$$M_A^{io} : Comm(\Sigma) \to (State(A) \to State(A))$$

where, for $S \in Comm(\Sigma)$ the purpose of the function

$$M_A^{io}(S) : State(A) \to State(A)$$

is, for $\sigma \in State(A)$

$$M_A^{io}(S)(\sigma) = \text{the final state, if such a state exists, on executing program } S \text{ on initial state } \sigma.$$

The definition is constructed by induction on the syntactic structure of a program S.

Base Case There are two base cases.

Identity Do nothing

$$M_A^{io}(\mathbf{skip})(\sigma) = \sigma.$$

Assignment Update a variable with evaluation of an expression

$$M_A^{io}(x{:=}e)(\sigma) = \sigma[\,V_A(e)(\sigma)/x].$$

Induction Step There are three cases.

Composition Execute S_1 and then S_2

$$M_A^{io}(S_1;S_2)(\sigma) \simeq M_A^{io}(S_2)(M_A^{io}(S_1)(\sigma)).$$

Conditional Choose to execute S_1 or S_2 according to test b

$$M_A^{io}(\textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi})(\sigma) = \begin{cases} M_A^{io}(S_1)(\sigma) & \text{if } W_A(b)(\sigma) = tt; \\ M_A^{io}(S_2)(\sigma) & \text{if } W_A(b)(\sigma) = f\!f. \end{cases}$$

Iteration Repeatedly execute S_0 until b is false. We define the semantics of the **while** command in two cases, depending upon whether or not a computation exits the **while** loop.

Termination Suppose the computation exits the **while** construct and halts.

$$M_A^{io}(\textbf{while}\ b\ \textbf{do}\ S_0\ \textbf{od})(\sigma) \downarrow\ \&\ M_A^{io}(\textbf{while}\ b\ \textbf{do}\ S_0\ \textbf{od})(\sigma) = \tau$$

if, and only if, there exists $n \geq 0$ and a sequence of states

$$\sigma_0, \sigma_1, \ldots, \sigma_n$$

such that

Initial state	$\sigma_0 = \sigma$
Final state	$\sigma_n = \tau$
Iteration	$M^{io}_A(S_0)(\sigma_{i-1}) \downarrow$ & $M^{io}_A(S_0)(\sigma_{i-1}) = \sigma_i$ for $1 \leq i \leq n$
Continuity	$W_A(b)(\sigma_i) = tt$ for $1 \leq i \leq n$
Exit	$W_A(b)(\sigma_n) = ff$

Non-termination Otherwise, suppose that the computation does not exit the **while** construct. This means there is no such finite sequence and $M_A^{io}(\textbf{while } b \textbf{ do } S_0 \textbf{ od})(\sigma)$ is undefined, which we denote by

$$M_A^{io}(\textbf{while } b \textbf{ do } S_0 \textbf{ od})(\sigma) \uparrow .$$

Examples Let

$$\sigma_{real}(x) = \mathit{3.14} \quad \text{and} \quad \sigma_{real}(y) = \pi.$$

Then

$$\begin{aligned} M_A^{io}(y{:=}x)(\sigma) &= \sigma[V_A(x)(\sigma)/y] \\ &= \sigma[\sigma_{real}(x)/y] \\ &= \sigma[\mathit{3.14}/y]. \end{aligned}$$

Let

$$\sigma(\mathsf{x}) = \pi.$$

Then

$$M_A^{io}(\textbf{while } \mathsf{x} > 0 \textbf{ do } \mathsf{x}{:=}\mathsf{x} + 1 \textbf{ od})(\sigma) = \perp$$

because the evaluation of the Boolean expression will always be true, so leading to a non-convergent computation sequence

$$\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_n, \ldots$$

in which $\sigma_n(x) = \pi + n$.

However, if we take the initial state σ, over which we evaluate S to have

$$\sigma_{real}(\mathsf{x}) = -\mathit{1}$$

then

$$M_A^{io}(\textbf{while } \mathsf{x} > 0 \textbf{ do } \mathsf{x}{:=}\mathsf{x} + 1 \textbf{ od})(\sigma) \downarrow \sigma$$

as

$$W_A(\mathsf{x} > 0)(\sigma) = \mathit{ff}$$

giving a computation sequence of one state $\sigma_{\mathit{0}} = \sigma$.

Statements and Commands: Second Definition using Recursion

An alternative approach is to develop an equational definition for the case of the **while** statement. We expect that the statement

$$S \equiv \textbf{while}\ b\ \textbf{do}\ S_0\ \textbf{od}$$

has the same effect on a state as the statement

$$S' \equiv \textbf{if}\ b\ \textbf{then}\ S_0; \textbf{while}\ b\ \textbf{do}\ S_0\ \textbf{od}\ \textbf{else}\ \textbf{skip}\ \textbf{fi}$$

which unfolds the first stage in the **while** loop.

Now both statements S and S' are valid **while** programs and hence have a formal input-output semantics *according to the first definition*. The input-output semantics of the first definition are the same:

Lemma (Semantics of unfolded while loops) *For any* $\sigma \in State(A)$,

$$M_A^{io}(\textbf{while } b \textbf{ do } S_0 \textbf{ od})(\sigma)$$
$$\simeq M_A^{io}(\textbf{if } b \textbf{ then } S_0; \textbf{ while } b \textbf{ do } S_0 \textbf{ od else skip fi})(\sigma)$$

This recursive definition provides for each S an equation that $M_A^{rec}(S)$ must satisfy.

$$\begin{aligned} M_A^{rec} &: \ Comm(\Sigma) \rightarrow (State(A) \rightarrow State(A)) \\ M_A^{rec}(S)(\sigma) &= \text{the final state, if such a state exists, on executing} \\ &\quad\ \text{program } S \text{ on initial state } \sigma. \end{aligned}$$

$$
\begin{aligned}
M_A^{rec}(\textsf{\textbf{skip}})(\sigma) &= \sigma \\
M_A^{rec}(x{:=}e)(\sigma) &= \sigma[V_A(e)(\sigma)/x] \\
M_A^{rec}(S_1;S_2)(\sigma) &\simeq M_A^{rec}(S_2)(M_A^{rec}(S_1)(\sigma))
\end{aligned}
$$

$$
M_A^{rec}(\textsf{\textbf{if}}\ b\ \textsf{\textbf{then}}\ S_1\ \textsf{\textbf{else}}\ S_2\ \textsf{\textbf{fi}})(\sigma)
= \begin{cases} M_A^{rec}(S_1)(\sigma) & \text{if } W_A(b)(\sigma) = tt; \\ M_A^{rec}(S_2)(\sigma) & \text{if } W_A(b)(\sigma) = f\!f. \end{cases}
$$

$$
M_A^{rec}(\textsf{\textbf{while}}\ b\ \textsf{\textbf{do}}\ S_0\ \textsf{\textbf{od}})(\sigma)
= \begin{cases} M_A^{rec}(\textsf{\textbf{while}}\ b\ \textsf{\textbf{do}}\ S_0\ \textsf{\textbf{od}})(M_A^{rec}(S_0)(\sigma)) \\ \quad \text{if } W_A(b)(\sigma) = tt; \\ \sigma \ \text{ if } W_A(b)(\sigma) = f\!f. \end{cases}
$$

By the semantics of unfolded **while** loops Lemma we know that the state transformer M_A^{io} satisfies the equations generalised from S. However,

(i) How many state transformers, in addition to $M_A^{io}(S)$ satisfy the equations?

(ii) Are there extra properties that allow us to characterise the function $M_A^{io}(S)$ as a unique solution of the equations?

Adding Data Types to Programming Languages

We have developed a formal definition of the syntax and semantics for the simple programming language

$$WP(\Sigma)$$

of **while** programs that compute over an abstract data type with signature Σ.

To compute over and implement the data type, we chose a Σ-algebra

$$A$$

and defined the input-output semantics

$$M_A^{io}(S) : State(A) \to State(A)$$

of every program $S \in Comm(\Sigma)$ over A.

Suppose we want to enhance the power of WP by adding some constructs, such as

(i) dynamic arrays, or

(ii) infinite streams.

This is trivial given our methods. As we have emphasised repeatedly, we have solved the problem for **while** programming over *any* algebra A.

Adding Dynamic Arrays

For any Σ-algebra A we can construct the algebra

$$A_{Array}$$

with signature Σ_{Array} of dynamic arrays over A.

So, given $WP(\Sigma)$, we can add dynamic arrays to our **while** programming language over Σ simply by forming the language

$$WP(\Sigma_{Array}).$$

We can obtain its semantics by applying our input-output model to Σ_{Array}-algebra A_{Array}.

Adding Infinite Streams

For any Σ-algebra A we can construct the algebra

$$A_{Stream}$$

with signature Σ_{Stream} of infinite streams over A.

So, given $WP(\Sigma)$, we can add infinite streams to our **while** programming language over Σ simply by forming the language

$$WP(\Sigma_{Stream}).$$

We can then obtain its semantics by applying our input-output semantics model to Σ_{Stream}-algebra A_{Stream}.