A 20-20 View of Ada
An Evolutionary Perspective

Tucker Taft
AdaCore

Ada-Europe 2019
Warsaw, Poland
June 2019

borrowed heavily from ...

AdaCore
TECH DAYS

Ada 2020 Update

S. Tucker Taft
November 15, 2017

AdaCore
The GNAT Pro Company

... some images from ...

Ada83 to Ada2012—Lessons Learned Over 30 Years of Language Design

John Barnes
John Barnes Informatics

Tucker Taft
AdaCore Inc

Portland, OR

October 2014

AdaCore
The GNAT Pro Company

… and a bit from …

Safe Parallel Language Extensions for Ada 202X

Tucker Taft, AdaCore

Co-Authors of HILT 2014 paper:
Brad Moore, Luís Miguel Pinho, Stephen Michell

June 2014

Ada 2020 High-Level Story

- **Make Ada a great language for parallel programming**
- **Other enhancements that build on Ada's existing strengths:**
  - **Safety and Security**
  - **Contract-Based Programming**
  - **Expressivity (particularly when it furthers the previous goals)**

Adding Support for Parallel Programming

Concurrent programming

Multiple computations

One or more workers

Often need to synchronize between computations

Parallel programming

One (or more) large computation(s)

Many workers

Synchronization typically only for work split/join

Adding Support for Parallel Programming

Concurrent programming

- Ada has great building blocks for concurrent programming
- Tasks, rendezvous, protected objects

Parallel programming

- Nothing currently built-in
- Although concurrent building blocks can be used, they’re very heavyweight

Ada 2020 Parallel Programming Goals

- Make it easy and safe to write parallel algorithms
- Hide the housekeeping of dispatch/scheduling/data collection
- Allow the compiler to choose among heterogeneous processors
  - N threads/processors, GPU, coprocessors, etc..
- Have the compiler detect and disallow data races

A reminder why this is important…
The 2005 *Right Turn* in Single-Processor Performance (14 years ago)

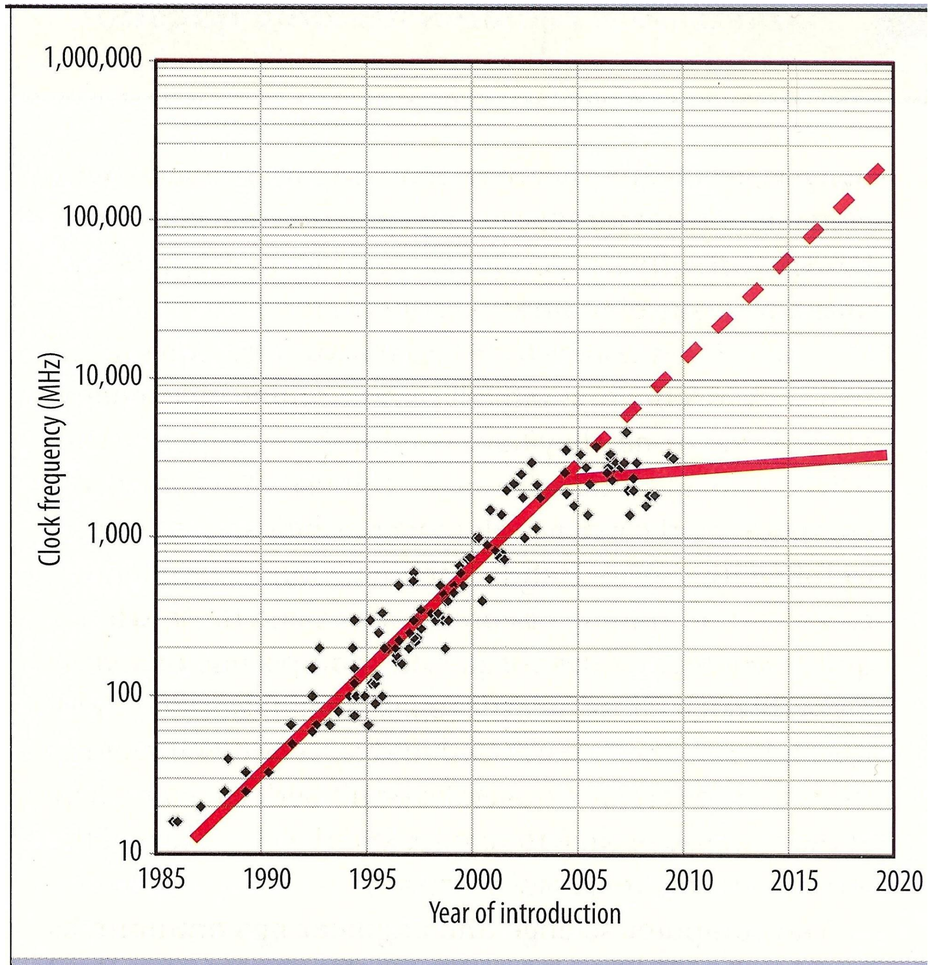


Figure 2. Historical growth in single-processor performance and a forecast of processor performance to 2020, based on the ITRS roadmap. A dashed line represents expectations if single-processor performance had continued its historical trend.

Courtesy IEEE Computer, January 2011, page 33.

Parallel Loops (2017)

```
parallel for I in 1 .. 1_000 loop
   A(I) := B(I) + C(I);
end loop;

parallel for Elem of Arr loop
   Elem := Elem * 2;
end loop;
```

Parallel Loops (202X)

```
parallel (2*Num_CPUs) -- Specify max level of parallelism
for I in 1 .. 1_000 loop
   A(I) := B(I) + C(I);
end loop;

parallel (Ck in Partial_Sum'Range)    --  A named chunk index
for Elem of Arr loop
   Elem := Elem * 2;
   Partial_Sum(Ck) := @ + Elem ** 2;  --  Manual reduction
end loop;
Sum := Partial_Sum'Reduce("+", 0.0);  --  Final reduction
```

Parallel Block (2014)

```
 parallel
     sequence_of_statements
{and
     sequence_of_statements}
 end parallel;
```

From “Gang of 4” HILT 2014 paper:
Each alternative is an (explicitly specified) *“parallelism opportunity”* (POp) where the compiler may create a *tasklet,* which can be executed by an *execution server* while still running under the context of the enclosing *task* (same task ‘Identity, attributes, etc.). Compiler will complain if any data races or blocking are possible (using Global and Potentially_Blocking aspect information).

Parallel Block (202X)

```
parallel do
    handled_sequence_of_statements
{and
    handled_sequence_of_statements}
end do;
```

From Ada 202x draft manual:
Each handled_sequence_of_statements represents a separate logical thread of control that proceeds independently and concurrently. The parallel_block_statement is complete once every one of the handled_sequence_of_statements has completed, either by reaching the end of its execution, or due to a transfer of control out of the construct by one of the handled_sequence_of_statements (see 5.1).

- *Parallel block is important for divide-and-conquer algorithms*
  - *such as sorting and searching*
  - *equivalent to parallel loop around a “case” statement*

Map/Reduce Iterators (2017)

```
--  A reduction expression to calculate the sum of elements of an array
Result : Integer := (for Element of Arr => <0> + Element)

--  A reduction expression to create an unbounded string
--  containing the alphabet
Alphabet : Unbounded_String
  := (for Letter in 'A' .. 'Z' => <Null_Unbounded_String> & Letter)

--  A reduction expression to determine how many
--  people in a database are 30-something
ThirtySomethings : constant Natural
  := (for P of Personnel => <0> + (if Age(P) > 30 then 1 else 0));
```

Map/Reduce Iterators (202X)

```
--  A reduction expression to calculate the sum of elements of an array
Result : Integer := [for Element of Arr => Element]'Reduce("+", 0);

--  A reduction expression to create an unbounded string
--  containing the alphabet
Alphabet : Unbounded_String
  := [for Letter in 'A' .. 'Z' => Letter]'Reduce("&", Null_Unbounded_String, "&");

--  A reduction expression to determine how many
--  people in a database are 30-something
ThirtySomethings : constant Natural
  := [for P of Personnel => (if Age(P) > 30 then 1 else 0)]'Reduce("+", 0);
```

Global contracts from SPARK (2017)

used for data race detection

```
Global => in out all -- default for non-pure pkgs
Global => null       -- default for pure packages

-- Explicitly identified globals with modes
Global => (in     P1.A, P2.B,
           in out P1.C,
           out    P1.D, P2.E)

-- Pkg data, access collection, task/protected/atomic
Global => in out P3                -- pkg P3 data
Global => in out P1.Acc_Type       -- acc type
Global => in out synchronized      -- prot/atomic
```

Global contracts from SPARK (202X)

used for data race detection

```
Global => in out all -- default for non-pure pkgs
Global => null       -- default for pure packages

-- Explicitly identified globals with modes
Global => (in     P1.A, P2.B,
           in out P1.C,
           out    P1.D, P2.E)

-- Pkg data, access collection, task/protected/atomic
Global => in out private of P3     -- pkg P3 data
Global => in out P1.Acc_Type       -- acc type
Global => synchronized in out all -- prot/atomic
```

Nonblocking contract
used for deadlock detection

- **Ada 202X Nonblocking aspect**

```
-- apply to one subprogram
procedure Suspend_Until_True
  (S : in out Suspension_Object)
    with Nonblocking => False;

-- apply to an entire package
package Ada.Characters.Handling
    with Nonblocking => True is …
```

- **Similar to “queued” qualifier in ParaSail**

Ada 202x Syntactic Building Blocks for Parallelism

Ada 202X Building Blocks -- Iterators

- ***Programmers Prefer Iterators***
  - Looping semantics very visible -- no mystery
  - Ada 2012 iterators made containers significantly more useful
    - E.g. AdaCore tool written in 2013 makes heavy use of iterators

- ***In Ada 202X, we build on iterators***
  - **For array aggregates defined with "iterated component association":**
    - X : Int_Array := (**for** I **in** 1 .. N => I**2)
  - **For aggregates defined by iterating over a container:**
    - Y : Int_Array := (**for** E **of** C => E);
  - **For "procedural iterators":**
    - Loop body becomes local anonymous procedure passed into existing iterator procedures:
      - such as Maps.Iterate and Environment_Variables.Iterate
  - **For reduction expressions** (see earlier examples)

Ada 202X Building Blocks -- Filters

- **Iterators sometimes generate too many values**
  - **Use filter to reduce to values of interest**
  - **for** iterator **when** condition **...**

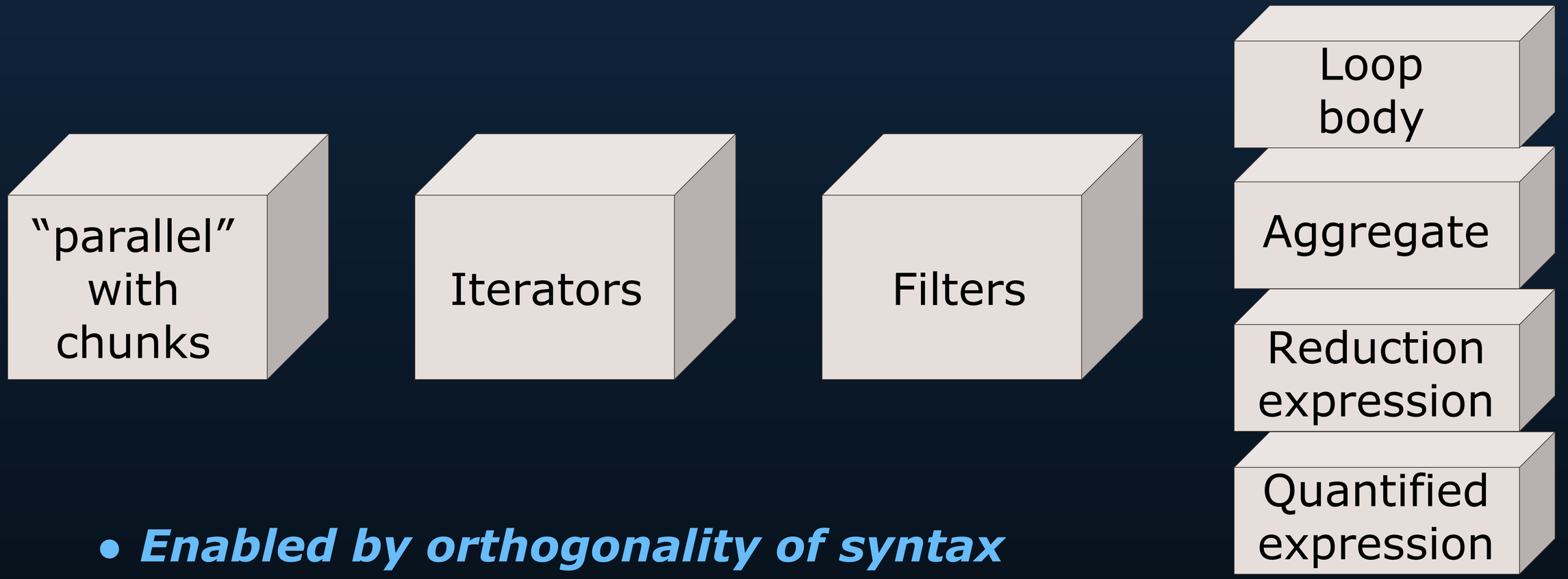
- **Filters can be used in various kinds of iterators:**
  - **For aggregates defined by iterating over a container:**
    - Odds : Int_Array := (**for** E **of** C **when** E **mod** 2 = 1 => E);
  - **For procedural iterators:**
    - **for** (Name, Value) **of** Environment_Variables.Iterate
      when Name(Name'First) /= "_" **loop**
        Put_Line (Name & " => " & Value);
      end loop;
  - **For reduction expressions:**
    - [**for** P **of** Personnel **when** Age(P) > 30 => 1]'Reduce("+", 0);

Ada 202X Building Blocks -- "parallel"

- **Iterators can generate many values**
  - **Use "parallel" to spawn multiple logical threads of control**
    - **parallel ...** *-- uses default amount of "chunking"*
    - **parallel** (Num_Chunks) **...** *-- specify a max number of chunks*
    - **parallel** (Chunk **in** 1..Num_Chunks) **...** *-- specify a chunk parameter*

- **"parallel" can be used with various kinds of iterators:**
  - **For iterating over a large range:**
    - **parallel** (Chunk **in** 1 .. Num_Chunks) *-- named chunk parameter*
      for I **in** Arr'Range loop
        . . . *-- possibly do other stuff*
        Partial_Sum(Chunk) := @ + Arr(I); *-- accumulator for each chunk*
      end loop;
      return Partial_Sum'Reduce("+", 0.0); *-- final reduction*
  - **For large reduction expressions over container iterator:**
    - [**parallel for** P **of** Personnel **when** Age(P) > 30 => 1]'Reduce("+", 0);
      - User defined "split" iterators for containers *-- like Java's "spliterators"*
- **Data race conflict checks provided at three levels -- All, Known, None**

Ada 202x uses “building block” approach



- ***Enabled by orthogonality of syntax***
- ***Eases use and readability relative to large library or multiple pragmas***
- ***Portable concepts that can be mapped to diverse targets***
- ***Compile-time conflict checking for safety***

Relevance of OpenMP & friends

- **OpenMP 1.0 -- 1997, OpenMP ARB**
  - *Heavy weight threads, SPMD model*
- **CUDA -- 2007, NVIDIA**
  - *NVIDIA GPUs, explicit separated "kernel" code*
- **OpenCL -- 2008, Apple, Khronos**
  - *Most GPUs, explicit separated "kernel"s*
- **OpenMP 3.0 -- 2008, OpenMP ARB**
  - *Lighter weight "tasks" with work sharing*
- **OpenACC -- 2011, Cray, NVIDIA, PGI**
  - *Many GPUs, no separate "kernel"*
  - *Extracts GPU "kernel" code from for-loop*
- **OpenMP 4.0 -- 2013, OpenMP ARB**
  - *Adds "target" devices, begins to subsume OpenACC*
- **OpenMP 5.0 -- 2018, OpenMP ARB**
  - *Largely subsumes OpenACC, and OpenCL to lesser extent*

OpenMP Evolution

- **OpenMP originally designed in 1997**
  - **Initially supported only heavy-weight "threads"**
    - *mapped generally to "kernel" threads*
    - *analogous to Ada "tasks"*
  - **Thread ID used explicitly to compute what part of data to manipulate**
    - *SPMD -- "Single Program, Multiple Data" model*
  - **Programs had no visible structure that matched computation being performed**
    - *Pragmas used heavily to provide implicit structure*
    - *No data race checking provided*

- **OpenMP evolved over time; OpenMP 5.0 is recently released (Nov 2018)**
  - *Early features mostly supplanted by newer notions based on lighter-weight "tasks" and work sharing.*
  - *Supports parallel loops of a specific structure (pragma + pattern)*
  - *Supports parallel blocks using an explicit "task" pragma*
  - *Incorporated OpenACC-like support for "targets" such as GPUs*
  - *Preferred over OpenCL or CUDA because can debug parallel algorithm on host before inserting "target" directives*
  - *Still no data race checking in most implementations*
    - *Explicit dependence annotations could eventually enable more checking*

Mapping Ada 202X to OpenMP & friends

- ***For Ada 202X mapping, we will generally use newer OpenMP features***
  - **Rely on Ada 202x language syntax for high-level parallel algorithm structure**
    - including correctness and highest level tuning (e.g. chunking)
  - **Rely on pragmas, aspects, and/or library calls for target-specific tuning:**
    - Controlling total number of heavy-weight threads
    - Data flushing and caching
    - Mapping to target devices

- ***Examples of specific mappings:***
  - **Parallel region establishes initial number of (heavyweight) threads**
    - Generally will create one region per Ada program
    - Want to minimize creating and releasing multiple heavyweight threads
  - **For parallel block, tasks are generated**
    - a “single” construct followed by two or more “task” pragmas (or API calls)
    - awaited at a “taskwait”
  - **For parallel loop:**
    - “parallel for” or “taskloop” pragma/API used for loops that match for-loop “pattern” supported by OpenMP
    - tasks spawned explicitly to handle other Ada 202X parallel loops, such as those for parallel container iterators, with explicit “taskwait”

Reprise:
Ada 2020 Parallel Programming Goals

- Make it easy and safe to write parallel algorithms
- Hide the housekeeping of dispatch/scheduling/data collection
- Allow the compiler to choose among heterogeneous processors
  - N threads/processors, GPU, coprocessors, etc..
- Have the compiler detect and disallow data races

Ada 2020 Expressivity Features

- Allow user to express their intent with less boilerplate
- More declarative fashion of doing things:
  - Usable in contracts
  - Smaller bug surface

Ada 2020 Expressivity Features (2017)

- **Delta aggregate notation: Update only part of a data structure**

```
Tax_Day : Date := (Today with delta Day => 15, Month => April);
```

- **Array aggregates defined by an Iterator**

```
Squares : array (Positive range <>) of Integer := (for I in 1 .. 256 => I ** 2);
```

- **Aggregates for containers (can be combined with previous features)**

```
package Maps is new Ada.Containers.Hashed_Maps (Integer, String, ..);

M  : Maps.Map := (1 => "Hello", 2 => "World");
```

Ada 2020 Expressivity Features (202X)

- **Delta aggregate notation: Update only part of a data structure**

```
Tax_Day : Date := (Today with delta Day => 15, Month => April);
```

- **Array aggregates defined by an Iterator**

```
Squares : array (Positive range <>) of Integer := (for I in 1 .. 256 => I ** 2);
```

- **Aggregates for containers (can be combined with previous features)**

```
package Maps is new Ada.Containers.Hashed_Maps (Integer, String, ...);
package String_Sets is new Ada.Containers.Hashed_Sets (String, ...);

  M  : Maps.Map := [1 => "Hello", 2 => "World"];
  S  : String_Sets.Set := [];  --  Empty set
begin
  S := [“Hello”];              --  Singleton set
```

- **Declare expressions**

```
with Post => (declare M renames Integer’Max(X, Y); begin F’Result = 2*M / (M-1))
```

Other Ada 202X Significant Changes

- Pre and Post for access-to-subprogram types and for generic formals
- Default_Initial_Condition to specify state after default initialization of a private type
- Pre, Post, Nonblocking, Global used to specify container packages
- Stable view for containers to support more efficient iteration
- Static expression functions
- The Image attribute for nonscalar types (arrays, records, etc.)
- User-specifiable attribute Put_Image provides user-defined Image
- User-defined Integer_Literal, Real_Literal, and String_Literal aspects.
- Arbitrary-precision integer and real arithmetic
- The Jorvik profile for lower criticality hard-real time systems

Ada 202X
Prototyping and Evaluation
(cf. Ada 80 Test & Eval)

Example Issues:
CPU vs. GPU vs. OpenMP focus
Race Condition and Deadlock Checking
Syntax vs. Pragmas vs. Library
Overall Ease of Understanding
Getting the Details Right

Ada 202X Feedback Time!

- Importance of supporting lightweight parallelism in Ada 202X
  - 1 = Not important, 5 = Very important

- Which is likely more important for Ada 202X users:
  - multicore CPUs
  - GPUs
  - no difference

- Importance of Race Condition and Deadlock Checking
  - 1 = Not important, 5 = Very important

- Favored approach to lightweight parallelism for Ada 202X
  - Syntax
  - Pragmas
  - Library

Ada 202X Feedback Time!

- **Array aggregates defined by an Iterator**

```
Squares : array (Positive range <>) of Integer := (for I in 1 .. 256 => I ** 2);
```

- **Aggregates for containers** (can be combined with previous features)

```
package Maps is new Ada.Containers.Hashed_Maps (Integer, String, ...);
package String_Sets is new Ada.Containers.Hashed_Sets (String, ...);

  M  : Maps.Map := [1 => "Hello", 2 => "World"];
  S  : String_Sets.Set := [];  --  Empty set
begin
  S := [“Hello”];              --  Singleton set
```

1. **Use [...] for container aggregates only**
2. **Use [...] for container aggregates, and allow for array aggregates**
3. **Use [...] for container aggregates, and allow for any aggregate**
4. **Don’t use [...] for container aggregates**
   a. **empty and singletons should use some other special syntax**

Whither Ada 2099?

Ada 83

- **Rock Solid Abstraction Capability**
  - Packages and Private Types
  - Very Strongly Distinguished Numeric Types
- **Completely Static Language**
  - No Type Extension
  - No Procedure Parameters
  - No Runtime Polymorphism
  - Case Statements Rule the World

Ada 95

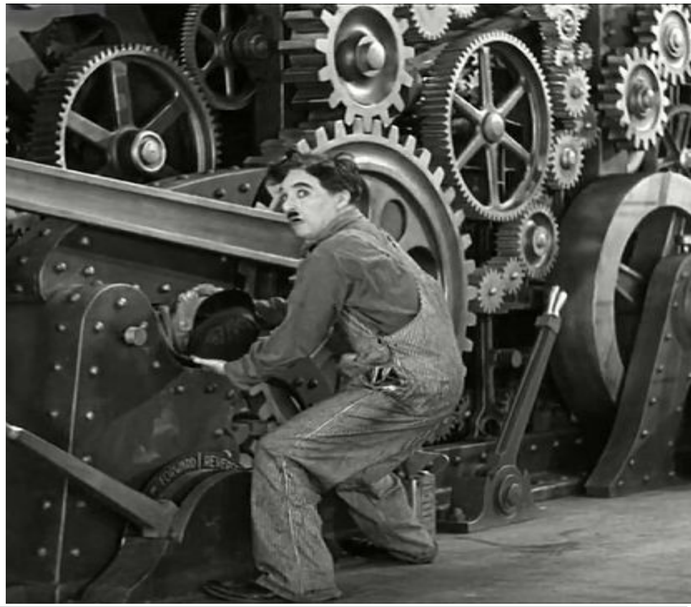
- **A Radical New Style – Very Dynamic**
  - Case Statements Considered Harmful
  - Inheritance and Polymorphism are the new Style
- **But Ada 95 was a bit spikey**
- **Some features not fully integrated**
  - *OO and Tasking don't play together well*
  - *Generics and OO are awkward partners*
- **No notion of Abstract Interfaces**
- **Relatively Low-Level Standard Libraries**

Ada 2005

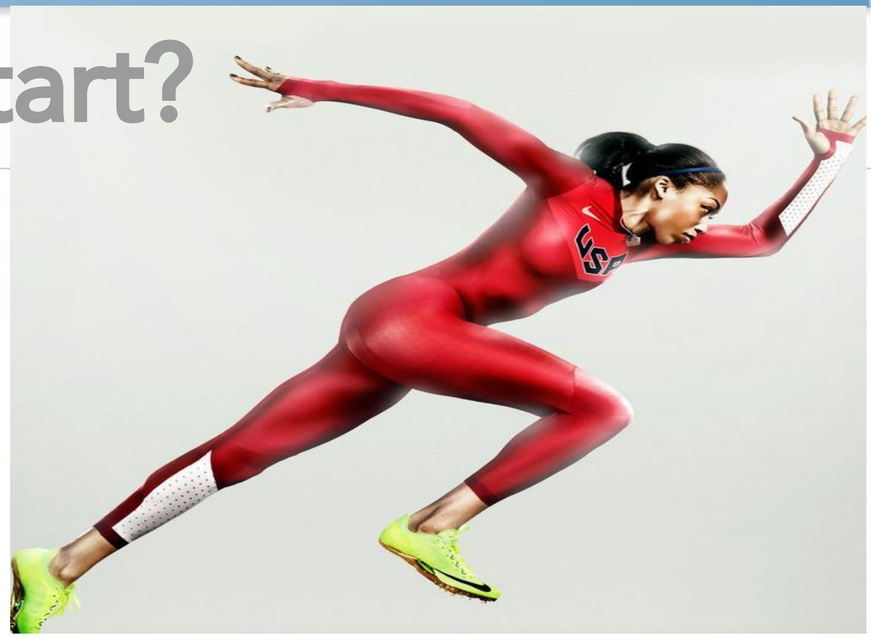
- **Integrated OO and Tasking – Far out**
- **Rounded off the spikey corners of Ada 95**
- **Created a Library of Containers**
  - Lists, Vectors, Sets, Maps
  - Encapsulate the Complexity
  - Raise the Level of Abstraction
  - But Containers Are A Bit of a Pain to Use
- **But not very exciting, no new killer apps**
- **Still Haven't Addressed Awkward Generic/OO Partnership**
  - Each generic instance represents a completely separate type hierarchy

Ada 2012

- **Don’t mess with Ada 2012**
- **Enforces Contract-Based Programming**
- **Gives Programmers More Power**
  - Conditional Expressions
  - Quantified Expressions
  - High-Level Container Iterators
- **But Elegant High-Level Features Depend on Ever-Expanding Complexity Below**
  - Dynamic object lifetime checks
  - Tampering Checks
  - Storage Subpools
  - Aliased Parameters
- **Generics Remain Too Heavyweight**
  - And not smoothly integrated into type hierarchy



Ada 202x -- Ada 2099 A New Start?



- Multicore revolution is a chance to rethink some basic assumptions
- Safety Through Simplification a la SPARK
  - Alias-free Pointers
  - Declared Side-Effects
  - Absence of Runtime Exceptions (AoRTE)
  - Generics and OO Integrated Smoothly
  - Syntactic Sugar Provides Uniform High-Level Abstractions
- Lightweight Safe Parallelism for all Iterators
- Still Looks and Feels Familiar while Reducing Complexity and Gaining Safety

Beam me up!