A Programming Model for Sustainable Software (Technical Report)

Haitao Steve Zhu Chaoren Lin Yu David Liu
†SUNY Binghamton, Binghamton, NY 13902, USA, Email: {hzhu1,clin18,davidL}@binghamton.edu

***Abstract*—This paper presents a novel energy-aware and temperature-aware programming model with first-class support for *sustainability*. A program written in the new language, named Eco, may adaptively adjusts its own behaviors to stay on a given (energy or temperature) budget, avoiding both deficit that would lead to battery drain or CPU overheating, and surplus that could have been used to improve the quality of results. Sustainability management in Eco is captured as a form of supply and demand matching, and the language runtime consistently maintains the equilibrium between supply and demand. Among the efforts of energy-adaptive and temperature-adaptive systems, Eco is distinctive in its role in bridging the programmer and the underlying system, and in particular, bringing both programmer knowledge and application-specific traits into energy optimization. Through a number of intuitive programming abstractions, Eco reduces challenging issues in this domain — such as workload characterization and decision making in adaptation — to simple programming tasks, ultimately offering fine-grained, programmable, and declarative sustainability to energy-efficient computing. Eco is an minimal extension to Java, and has been implemented as an open-source compiler. We validate the usefulness of Eco by upgrading real-world Java applications with energy awareness and temperature awareness.**

I. Introduction

Two trajectories appear to shape up the evolution of modern computing platforms: they become increasingly "nimble," or increasingly "mighty." "Nimble" computers—such as RFIDs, wearable electronics, smartphones, tablets, and laptops—often operate on battery with a limited budget. "Mighty" computers—represented by data centers, cloud servers, and many-core clusters—not only consume a large amount of energy, but also face grave reliability concerns due to excessive heat dissipation, with cooling taking up to 35% of their operational costs [3]. Effective energy and thermal management is a critical goal demanding solutions from all layers of the compute stack. In recent years, there is a growing interest in addressing these challenges from the perspective of application software (*e.g.*, [16], [5], [21], [35], [25], [17], [33], [34], [29], [32], [20], [39], [4], [36], [10], where energy and thermal efficiency is pursued and achieved throughout the software lifecycle.

In this paper, we focus on an essential design goal in energy and thermal management: *sustainability*. A sustainable program is able to "stay on budget," neither yielding deficit nor surplus. In the context where "budget" refers to available battery — *e..g.*, 20% battery of a smartphone — sustainability

*The first and second authors have recently joined Google Inc. after completing the work at SUNY Binghamton.

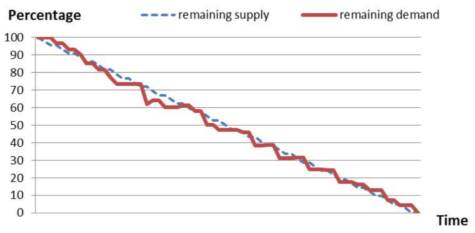

Fig. 1. A Sustainable Program Runtime

entails *energy awareness*: the sustainable program makes the best effort to not exceed 20% of battery usage (no deficit), while at the same time producing as high-quality results as possible (no surplus). In the context where "budget" refers to the allowance of temperature increase — *e.g.*, 10°C — sustainability entails *temperature awareness*: avoiding overheating while producing the best results. In essence, sustainability captures a recurring theme in energy and thermal management: balancing the trade-off between energy/thermal budget and the quality of computational results.

A. Eco *for Sustainable Programming*

Our concrete proposal is Eco, a novel and simple programming model to help developers construct sustainable software by design. As we look forward, we believe programmers have a unique role in application-level energy and thermal management, and the future of software on "nimble" and "mighty" computing platforms belongs to applications that neither "squander" nor "skimp" on resources.

The key insight of the Eco design is that sustainability can be achieved through the negotiation between *supply* and *demand*, ultimately maintaining the equilibrium of the two in a fashion similar to market economics. Syntactically, Eco is a minimal extension to Java, with two important programming abstractions:

- **Demand/Supply Characterization**: an Eco programmer may associate a fine-grained program component of interest (the "demand") with a budget (the "supply"), through a new construct called *sustainable blocks*
- **Adaptability Support**: an Eco programmer may define *alternative* application behaviors which may achieve the same application-specific goal but with different quality levels, through a principled and fine-grained construct called *first-class mode cases*

Throughout the execution of the code enclosed in the sustainable block, the Eco program runtime continuously monitors the change in demand and supply. As demonstrated in Figure 1, the sustainable execution should have the remaining demand (the solid line) closely matches the remaining supply (the dotted line). If demand far exceeds supply at a snapshot of the execution, the program adapts to program behaviors that may produce results in lower quality. If supply far exceeds demand, adaptation in the opposite fashion happens.

In a nutshell, the design scope of Eco subsumes programming abstractions to capture the essence of sustainable programming, compiler inference algorithms to automatically identify program points for monitoring and adaptation, and language runtime support to efficiently maintain the balance between demand and supply.

B. Contributions

To the best of our knowledge, Eco is the first programming model to provide first-class support for characterizing, negotiating, and maintaining budget in energy-aware and temperature-aware applications. This is also the first time fundamental concepts in market economics — supply and demand — are crystalized as programming abstractions. In the context of existing energy/thermal budget management systems [12], [38], [40], [9], [11], [13], [27]—most from VLSI, architecture, and OS communities—Eco is distinctive in its attempt to bridge application software with underlying systems, for at least three distinct reasons:

1) it elevates predominately system-level concepts such as energy budgeting and temperature control to the forefront of application software development, through supply characterization.
2) it taps the knowledge and insight of programmers to automate fine-grained energy and thermal management, through demand characterization.
3) it brings green-conscious programmers into the equation of energy and thermal optimization, through principled programmer-defined adaptability support.

Eco is a rigorously defined language [41], but more importantly, it is also a practical and expressive programming model for developing sustainable real-world applications. This paper places emphasis in highlighting Eco's role in green software engineering: how it empowers programmers with battery and temperature awareness, how it helps balance the trade-off between energy/thermal efficiency and quality of service, and what the recurring programming patterns in sustainable software are. Eco is developed as an open-source compiler:

`https://github.com/pl-eco/ECO`

In summary, this paper makes the following contributions:

- It introduces a simple programming model to provide fine-grained sustainability through supply/demand characterization, program transformations to automate and optimize sustainability management, and a program runtime design to allow for supply/demand negotiation.
- It provides first-class support to adaptive software. The design considers the consistency in adaptation, and seamlessly integrates adaptability and sustainability through compiler-directed automatic calibration and adaptation.
- It describes a set of common programming idioms, which may serve as a first-step toward understanding programming patterns in sustainable programming and constructing energy-aware and temperature-aware software.
- It demonstrates the feasibility and effectiveness of sustainable programming through extending real-world applications with the ability of battery awareness and temperature awareness, enabled by an open-source compiler.

```
mode { lo <: mid };
mode { mid <: hi };

class Raytracer {
  mcase<Res> res = {
    lo: new Res(640, 480);
    mid: new Res(800, 600);
    hi: new Res(1024, 768);
  };

  mcase<int> depth = {
    lo: 10;
    mid: 20;
    hi: 30;
  };

  Ray[][] getView(Camera camera, Res res) { ... }

  Image[] run(Camera camera, Model[] models) {
    Image[] images;
    int num, IMAX, JMAX, IMGS = 100;
    sustainable {
      for (num = 0; num < IMGS; num++) {
        uniform {
          Ray[][] view = getView(camera, res);
          Image image = new Image(res);
          IMAX = res.y;
          JMAX = res.x;
        }
        for (int i = 0; i < IMAX; i++) {
          for (int j = 0; j < JMAX; j++) {
            image.setColor(i, j,
              view[i][j].trace(models[num], depth)
            );
          }
        }
        images[num] = image;
      }
    }
    bsupply (0.2 * battery)
    demand (IMGS) -> (IMGS - num);
    return images;
  }
}
class Model {...}   // model to be rendered
class Camera {...}  // camera for rendering
class Image {...}   // 2D array of color pixels
class Ray {...}     // ray
class Res {...}     // resolution
```

Fig. 2. A Battery-Aware Raytracing Application

II. Sustainable Programming by Example

We use a simple raytracing example in Figure 2 to demonstrate the basic ideas of sustainable programming. Raytracing is a technique in rendering where an object in the form of a `Model` is rendered into an `Image`—a 2D array of colored pixels—based on the `Rays` cast from the `Camera` to the object. Eco is an extension to Java, with several notable features we highlight next.

a) Sustainable blocks: An energy-conscious programmer may declare the most expensive part of her program in *sustainable blocks*. In this example, the block encloses the ray tracing for `IMGS` number of images, as shown between Line 22 and Line 41. The programmer can set a budget for executing this code fragment through *supply characterization*: in Line 40, the budget (also known as *supply sum*) is set at 20% of the remaining battery. To help the language runtime correlate the change in battery over time and the progress of the program execution, Eco programmers are asked to provide a "progress indicator" through *demand characterization*. In this simplified example, the programmer just says that the *demand sum—i.e.*, the overall number of units of work—is `IMGS` (the number of images), and at a given moment during the sustainable block execution, the *demand gauge*—the indicator for the remaining number of units of work—is dependent on the loop index `num`, *i.e.*, through expression `IMGS - num`. Based on supply characterization and demand characterization, Eco introspects

- the proportion of remaining demand, *i.e.*, demand gauge relative to demand sum and
- the proportion of remaining supply, *i.e.*, remaining battery supply relative to overall battery supply

at various automatically determined "checkpoints" during the execution of the sustainable block, and deem the execution: (i) sustainable if the two are on similar levels; (ii) deficit if the former significantly exceeds the latter; (iii) surplus if the latter significantly exceeds the former.

b) Modes and First-Class Mode Cases: Whenever deficit or surplus happens, the program adjusts itself, guided by specifications of modes and first-class mode cases.

Borrowed from a previous work [10], modes are a discrete characterization of energy adaptation states. In Line 1 and Line 2, the program defines three modes, `lo`, `mid`, and `hi`, indicating the program may function under three different states of adaption. In Eco, modes form a total order, with `lo`/`mid`/`hi` in the example aligned with the intuition that the resulting code is likely to consume the least/average/most energy and produce results of the lowest/average/highest quality, respectively.

Modes matter in Eco because they play a central role in the design of *mode cases*. In a nutshell, a mode case is a data-centric way to define alternative behaviors in the presence of energy adaption. In the example, two mode cases are defined as fields: (i) the `res` field in Line 5 that says the `Raytracer` may use the resolution of 640×480 when program is executing the `lo` mode, 800×600 in the `mid` mode, and 1024×768 in the `hi` mode; (ii) the `depth` field in Line 11 that says the depth of `Raytracer`—a key parameter to control the quality of raytracing—may be either 10, 20, or 30, depending on the mode of the executing program. Mode cases in Eco are *first-class values*: they can be assigned to variables, passed around as method parameters, or stored in fields.

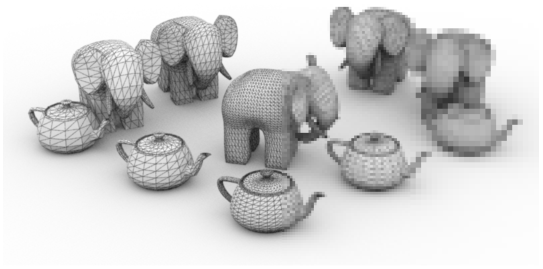

Fig. 3. Potentially Inconsistent Results (This image was produced by executing a variation of our `sunflow` benchmark — a real-world raytracing application detailed in Section VI — where the uniform block enclosing the image rendering is removed.)

To tie everything together, the executing sustainable block "downgrades" the mode of its execution whenever deficit happens—picking a precedent in the total order—and the mode case used in that block can be intuitively viewed as automatically reduced to the value associated with the new mode, a step we call *mode case selection*. For example, the mode case of `depth` under the execution of `mid` mode would be selected into 20. The scenario for surplus is similar, except that the sustainable block "upgrades" its mode. The described behavior here is aligned with our intuition of sustainability: the program adapts to the less energy-consuming behavior when it runs into a deficit and to the more energy-consuming behavior when it runs into a surplus.

c) Uniform Blocks: Coming with the flexibility of first-class mode cases is a design challenge: allowing for mode case selection at arbitrary time may yield results unacceptable to programmers. For instance, had we allowed the image resolution to change half way through raytracing an image, the program may produce a picture such as Figure 3, whose uneven rendering may unsettle (some) programmers.

An Eco programmer may choose to declare a block of code as *uniform*, as in Line 24 to Line 29. This precludes mode change from happening during the execution of said block. This consistency-preserving construct is reminiscent of the property of atomicity [15], [22], [19].

d) Battery-Awareness and Temperature Awareness: The example here is an instance of battery-aware programming. With minimal changes, the same program can turn into a temperature-aware one: by simply changing Line 40 to **tsupply**(10), the same sustainable block is to be executed with a (temperature) budget of 10°C, *i.e.* not raising CPU temperatures by more than 10°C. The similarity and symmetry of battery-aware programming and temperature-aware programming in Eco do not come as a surprise. They are unified under the same philosophy: just like the allowance of battery usage during a program execution can be viewed as a "budget," the allowance of temperature raise during a program execution can also be viewed as a "budget." As a consequence, demand/supply-style characterization and programming are applicable to both goals, and the unified programmer view is an attractive trait to improve programmer productivity.

Indeed, the unification of battery-aware programming and temperature-aware programming goes deeper than syntactical similarities. As we shall see in the next section, the **bsupply** and **tsupply** constructs are both syntactical sugars of a more general programming abstraction, and the two hence share the same semantic definition.

III. The Eco Design: Syntax and Runtime Support

A. Syntax Overview

Just like Java, an Eco program consists of a sequence of classes, with one containing the entry method `main`. In addition, an Eco program allows for top-level *mode declarations* in the form of **mode** m <: m. We require the <: relation defined through these declarations forms a total order. We further call the least element and greatest element in the total order the *least mode* and *greatest mode* respectively. Implicitly parameterized by these top-level mode declarations, unary operators $\uparrow$ and $\downarrow$ are standard successive and precedent operators over total order. Furthermore, we define $\uparrow \mathtt{m}$ as m if m is the greatest mode and $\downarrow \mathtt{m}$ as m if m is the least mode.

The core expressions of Eco are defined as follows:

$$\begin{array}{rcll} e & ::= & \textbf{uniform}\{e\} & \textit{expression} \\ & | & \{\mathtt{m}_1 : e_1; \ldots \mathtt{m}_n; e_n\}^\tau & \\ & | & \textbf{sustainable}\ \{e\}(c) & \\ & | & \textbf{select}\ e & \\ & | & \textbf{calib}\ e & \\ & | & \text{Java expressions} & \\ c & ::= & \langle e; e; e; e\rangle & \textit{characterizer} \\ \tau & ::= & \textbf{num}\ \mid\ \mathtt{X}\ \mid\ \textbf{mcase}\langle\tau\rangle & \textit{type} \end{array}$$

For simplicity, we treat statements as expressions. Expression **uniform**$\{e\}$ defines a uniform block whose body is e. We explain the rest of the syntax shortly. A type in Eco may take three forms: 1) a primitive type (*e.g.*, integer, float), which is abstractly represented as **num** for simplicity here; 2) an object type X, where X is the name of a class; 3) a type for the mode case value **mcase**$\langle\tau\rangle$, which says that the mode case has case members whose types are τ.

B. Runtime Behavior Overview

The unique runtime features of Eco center around the execution of a sustainable block. There, the (sum and remaining) supply and the (sum and remaining) demand are monitored through checkpoints, each of which we call a step of *supply/demand calibration*, or *calibration* for short. This leads to four interconnected questions at the center of Eco design:

- **Q1**: How to calibrate?
- **Q2**: How should the program adapt based on calibration (through mode case selection)?
- **Q3**: When to calibrate?
- **Q4**: When to select mode cases?

We answer **Q1** in Section III-C3 and **Q2** in Section III-D. To decouple **Q3** from **Q1**, and **Q4** from **Q2** for presentation purposes, we introduce two convenience expressions to make calibration point and mode case selection point explicit, **calib** e, and **select** e. The first expression says that calibration should be triggered after the evaluation of e, and the second expression says that the mode case represented by e should be selected. In other words, when we answer **Q1** and **Q2**, we circumvent the "when" questions by assuming they are explicitly "marked" in the program. In Section IV, we will introduce several program transformations to answer **Q3** and **Q4**, *i.e.*, automatically inserting **calib** e and **select** e expressions to the Eco source program.

C. Syntactical and Runtime Support for Sustainability

1) Syntax and Encodings: The general form of a sustainable block in Eco is as follows:

```
sustainable {
  ... // code in the sustainable block
}
supply (e1) -> (e2)
demand (e3) -> (e4)
```

where e_1, e_3, e_4 are the expressions for representing the *supply sum*, *demand sum*, *demand gauge* we informally introduced in Sec. II, respectively. The additional expression e_2 is called the *supply gauge*, an expression to indicate the remaining supply. In this presentation, we also abbreviate the sustainable block above as **sustainable** $\{\ldots\}(\langle e_1; e_2; e_3; e_4\rangle)$, and further call $\langle e_1; e_2; e_3; e_4\rangle$ the *characterizer* of the sustainable block.

Indeed, the sustainable blocks with **bsupply** and **tsupply** keywords introduced earlier are syntactic sugars, which can be encoded as in Fig. 4. As we shall see in Sec. V, Eco programmers can account for flexible forms of supply variability through supply gauge programming.

2) Sustainable Stack: A Eco program runtime maintains a key data structure called a *sustainability stack* (represented by *S*), or S-stack for short. The S-stack maintains the contexts of sustainable/uniform blocks: at a runtime state, each entry of this stack intuitively says that the current encountered program point is *dynamically scoped* in the sustainable/uniform block represented by the entry, with the top entry representing the most recently encountered scope. There are two important features to highlight from this design:

- Eco supports sustainability management across lexical scopes.
- Eco supports nested sustainable blocks, and nesting between sustainable blocks and uniform blocks.

The first support is a necessity for object-oriented languages: object references can escape lexical scope, and as a result, the program point of calibration may not be lexically enclosed in the sustainable/uniform blocks. In the technical report [41], we provide a detailed example to illustrate this technicality. The S-stack design flexibly enables sustainability management for all program points reachable from a sustainable/uniform block.

The support of nested blocks is important for two reasons. First, this facilitates modular programming, *e.g.*, in a scenario where the designer of a library API declares a sustainable block within the API method, and at the same time the client program using the API may declares its own (nesting) sustainable block. More importantly, nesting support also enables

$$
\begin{array}{rcl}
\textbf{battery} & \stackrel{\text{def}}{=} & \_b.\texttt{getB}() \\
\textbf{sustainable}\{e\}\ \textbf{bsupply}\ (e_1)\ \textbf{demand}\ (e_2)\text{->}\ (e'_2) & \stackrel{\text{def}}{=} & \textbf{num}\ \texttt{x} = \_b.\texttt{getB}(); \\
& & \textbf{sustainable}\{e\}\ \textbf{supply}\ (e_1)\text{->}\ (e_1 - \texttt{x} + \_b.\texttt{getB}())\ \textbf{demand}\ (e_2)\text{->}\ (e'_2) \\
\textbf{temperature} & \stackrel{\text{def}}{=} & \_t.\texttt{getT}() \\
\textbf{sustainable}\{e\}\ \textbf{tsupply}\ (e_1)\ \textbf{demand}\ (e_2)\text{->}\ (e'_2) & \stackrel{\text{def}}{=} & \textbf{num}\ \texttt{x} = \_t.\texttt{getT}(); \\
& & \textbf{sustainable}\{e\}\ \textbf{supply}\ (e_1)\text{->}\ (e_1 - \texttt{x} + \_t.\texttt{getT}())\ \textbf{demand}\ (e_2)\text{->}\ (e'_2)
\end{array}
$$

Fig. 4. Syntax Desugaring for Battery/Temperature-Aware Sustainable Programming (Let *_b* and *_t* variables of `BatteryManager` and `TempManager` types respectively. The `BatteryManager` class encapsulates the program logic for interacting with battery hardware, with one method `getB` to query the current battery level. The `TempManager` class encapsulates the program logic for interacting with temperature sensors , with one method `getT` to query the current temperature)

flexible sustainable programming where battery awareness and temperature awareness is required at the same time.

Structurally, each S-stack entry can take one of the two forms (1) a two-element tuple in the form of $(e : \texttt{m})$, where e is the characterizer expression of the sustainable block, and $\texttt{m}$ is the mode of the current sustainable block execution, called the *operational mode*. (2) a constant element $\perp$, indicating the entry representing a uniform block.

When the program bootstraps, we place an initial entry to the S-stack, whose operational mode is the greatest mode in the mode specification (a total order). This default treatment is aligned with our intuition that an expression not reached from any programmer-declared sustainable block should not subject to sustainability management, so the mode that represents the highest quality of service is used. Upon the entry of a programmer-declared sustainable block, an entry $(e : \texttt{m})$ is pushed to the stack, where e is the characterizer, and $\texttt{m}$ is initially the same as the "current" operational mode before the block is entered, where the notion of "current" is defined by the following function:

$$
\begin{array}{lcll}
\text{MNOW}(S) & \stackrel{\text{def}}{=} & \texttt{m} & \text{if } S = S', (e : \texttt{m}) \\
\text{MNOW}(S) & \stackrel{\text{def}}{=} & \text{MNOW}(S') & \text{if } S = S', \perp
\end{array}
$$

This design is useful in the presence of nested sustainable blocks, where the inner block starts operating at a mode sustainable for the outer block.

3) Sustainability Management as Supply/Demand Matching: Upon the evaluation of the **calib** expression, there are two possibilities:

- If any entry in the S-stack is $\perp$, the calibration is a no-op.
- Otherwise, *supply/demand matching* will be performed for all S-stack entries, with the top item being performed first. The detailed matching process will be explained next. The definition supports nesting, where sustainability management applies to all blocks, with the most immediate (inner) block first.

For each instance of supply/demand matching, the supply sum expression, the supply gauge expression, the demand sum expression, and the demand gauge expression will in turn be evaluated to a numerical value. Let us denote them as n_1, n'_1, n_2, n'_2 respectively. The matching between supply and demand is captured by the following predicate, where ϵ is a small positive numeric constant close to 0:

$$
\begin{array}{lcl}
\text{MATCH}(\langle n_1; n'_1; n_2; n'_2 \rangle) & \stackrel{\text{def}}{=} & |\frac{n'_1}{n_1} - \frac{n'_2}{n_2}| \leq \epsilon \\
\text{SURPLUS}(\langle n_1; n'_1; n_2; n'_2 \rangle) & \stackrel{\text{def}}{=} & \frac{n'_1}{n_1} - \frac{n'_2}{n_2} > \epsilon \\
\text{DEFICIT}(\langle n_1; n'_1; n_2; n'_2 \rangle) & \stackrel{\text{def}}{=} & \frac{n'_2}{n_2} - \frac{n'_1}{n_1} > \epsilon
\end{array}
$$

Eco language implementation selects $\epsilon = 0.1$ by default: the program is considered "sustainable" if the remaining proportions of supply and demand are within $\pm 10\%$ difference.

D. First-Class Mode Cases

Expression $\{\texttt{m}_1 : e_1; \ldots \texttt{m}_n; e_n\}^{\tau}$ represents a mode case, where each element $\texttt{m}_k : e_k$ (for $1 \leq k \leq n$) in the sequence is called a *case member*. At run time, the expression associated with each case member is evaluated at mode case definition time. As a result, each case member of a mode case holds a value at runtime.

First-class mode cases at run time are selected based on the operational mode. For now, to circumvent **Q4**, let us assume all mode case selection points are explicitly identified, through the **select** expression. The expression **select** e at run time evaluates to v_i following this definition:

$$
\{\texttt{m}_1 : v_1, \ldots, \texttt{m}_n : v_n\}/\texttt{m} \stackrel{\text{def}}{=} v_i
$$

where e evaluates to $\{\texttt{m}_1 : v_1, \ldots, \texttt{m}_n : v_n\}$, $\texttt{m}$ is the operational mode, and $\texttt{m}_i$ is the least mode among $\{\texttt{m}_1, \ldots, \texttt{m}_n\}$ that is equal to or greater than $\texttt{m}$ according to the total order defined by the program.

Recall that the program bootstraps with the greatest mode. This implies when a **select** expression appears outside of any sustainable block, the greatest mode is used. This aligns with our intuition that code not subject to sustainability management should not be approximated.

IV. The Eco Design: Compiler Support

In this section, we describe compiler support, primarily answering **Q3** and **Q4**.

A. Automated Calibration Insertion

We now define a compiler transformation to automatically identify and insert **calib** expressions. Several design choices are possible: (1) check periodically; (2) check whenever the supply gauge changes; (3) check whenever the demand gauge changes. Route (1) is *ad hoc*, especially when selecting a fixed "period" that fits all programs. For (2), the supply gauge in energy-aware programming is often associated with system-level variables (such as remaining battery) whose updates are often performed outside the program runtime and out of the control of programmers. This may lead to platform-dependent behaviors: the same program may calibrate at different rates on two computers with different battery drivers.

Eco follows route (3). Demand gauge is typically an expression formed with program variables, such as loop index in the example in Figure 2. The state change of these variables is visible and predictable to the programmer. For application-level energy management strategies—a family that Eco belongs to—we believe the programmer should have control on program behaviors related to energy awareness.

We now define the transformation process. The transformation is isomorphic except that expression e is transformed to **calib** e iff $\neg P_0(e) \wedge (P_1(e) \vee P_2(e))$, where:

$P_0(e)$: e can only be reached from uniform blocks

$P_1(e)$: e is assignment, and the left-hand side variable appears in the demand gauge of lexically enclosing sustainable blocks

$P_2(e)$: e is a field write expression, and said field is read in the demand gauge of any sustainable block that may reach e

The implementation of the three predicates has little novelty. P_1 is a trivial local analysis. Our Eco implemented P_0 and P_2 through a standard context-sensitive reachability analysis. Furthermore, observe that when e is reachable from both uniform blocks and non-uniform blocks, **calib** will be inserted. This is sound because its evaluation will become a no-op when reached from a uniform block ($\perp$ is in the S-stack).

B. Automated Mode Selection Insertion

The syntax we introduced earlier requires programmers to select mode cases explicitly through the **select** expression. For example, to assign a variable `y` of **mcase**$\langle$**int**$\rangle$ type to variable `x` of **int** type, programmers need to write `x` = **select** `y`. This can be an onerous task for programmers, especially for *incremental* programming, where programmers start with a program written in an existing language, and incrementally "try out" new language abstractions. An Eco programmer may start with a Java program, and alter a variable from, say, holding a `Res(1024, 768)` object to the mode case we used in Figure 2. If explicit **select** is needed, the programmer would have to manually insert a **select** expression for *every* occurrence of such variables in a sustainable block where the mode case is meant to be selected.

We define a simple compiler transformation to relieve the programmers of the burden of explicitly declaring **select** expressions. The key insight is that Eco as a strongly typed language provides sufficient type information to compare both the sending and the receiving ends of a data flow. For example, if a programmer writes `x` = `y` in the program, and variable `y` is of **mcase**$\langle$**int**$\rangle$ type and variable `x` is of **int** type, our compiler transformation algorithm will transform it into `x` = **select** `y`.

More generally, the transformation is hinged upon the definition of predicate $(\tau \nearrow \tau')\#(e \nearrow e')$, which says an expression e of type τ should be transformed to e' if it appears at a program point expecting an expression of type τ'. The predicate holds iff either of the two holds:

- $\tau << \tau'$
- $\tau << \tau'$ does not hold, and $\tau = \textbf{mcase}\langle\tau''\rangle$, and $(\tau'' \nearrow \tau')\#(e \nearrow e')$.

where the $<<$ relation is reflexive and transitive, with two additional rules. $\tau << \tau'$ iff (i) τ is a subclass of τ'; (ii) $\tau = \textbf{mcase}\langle\tau_0\rangle$ and $\tau' = \textbf{mcase}\langle\tau_0'\rangle$ and $\tau_0 << \tau_0'$.

Some examples should be sufficient to illustrate the ideas here:

$$(\textbf{mcase}\langle\textbf{num}\rangle \nearrow \textbf{num})\#(e \nearrow \textbf{select } e)$$
$$(\textbf{mcase}\langle\textbf{mcase}\langle\textbf{num}\rangle\rangle \nearrow \textbf{num})\#(e \nearrow \textbf{select } (\textbf{select } e))$$
$$(\textbf{mcase}\langle\texttt{X}\rangle \nearrow \texttt{Y})\#(e \nearrow \textbf{select } e) \quad \text{where } \texttt{X} \text{ is a subclass of } \texttt{Y}$$

Finally, the transformation is isomorphic, except that:

- **R1**: For every expression where (sub-)expression e flows to (sub-)expression e', where the type of e and e' are τ and τ' respectively, we transform the aforementioned expression to one the same as before, except that e is replaced with e'', and $(\tau \nearrow \tau')\#(e \nearrow e'')$. Here, the "flows-to" relation is the standard notion in data flow analysis.
- **R2**: For every expression where the (sub-)expression e appears in a program point that serves as an object target (such as the receiver of the method inovcation, or field read or write), we transform the aforementioned expression to one the same as before, except that e is replaced with e'', and $(\tau \nearrow \texttt{X})\#(e \nearrow e'')$, and $\text{ATOM}(\tau') = \texttt{X}$.

where $\text{ATOM}(\tau)$ is defined as τ, except that it is defined as $\text{ATOM}(\tau')$ if $\tau = \textbf{mcase}\langle\tau'\rangle$.

C. Type Checking

Type checking is defined over the post-translation programs where **calib** and **select** expressions are explicit. It is largely standard. The subtyping relation is defined to be identical as the $<<$ relation. For $\{\mathtt{m}_1 : e_1; \ldots \mathtt{m}_n; e_n\}^\tau$, we require the type of e_i for $1 \leq i \leq n$ be a subtype of the declared type τ.

V. Programming Idioms

We now summarize common idioms we encountered while programming in Eco. Broadly, the summary may serve as a first step toward understanding design patterns and micro-patterns of sustainable software.

A. Abstract Supply Units

The sugared syntax **bsupply** use battery capacity (mwh) as the unit of supply, and syntax **tsupply** uses temperature in Celsius. These units are not hardcoded in Eco. More generally, an Eco programmer may choose any unit intuitive and accurate for her sustainability management, as long as such values can be obtained from underlying OS and hardware. For example, popular battery-powered consumer computers (laptops and smartphones) typically come with power management modules that can report the remaining operating time (or remaining percentage). A sustainable block that should consume no longer than 20% of the remaining operating time can be programmed as:

```
sustainable {
   double startROT = 0.2 * getROT();
   ...
}
(< startROT; startROT − getROT(); ...; ... >)
```

where `getROT` queries the system battery management module for the remaining operating time.

In the real world, the suitable unit to report battery State-of-Charge (SoC) [14] is dependent on many factors, such as (i) the characteristics of batteries, *e.g.* lithium-based or nickel-based; (ii) the support of OS, *e.g.* ACPI-compliant systems and different vendor extensions (such as `BatteryService`[1] in Android); (iii) the estimation algorithms. The more user-friendly metrics—remaining percentage, or remaining operating time—are in fact derived metrics from lower-level battery status data *e.g.*, voltage level, current level, and coulomb counts [14]. Under the backdrop of this diverse and fast-changing landscape, the abstraction provided by Eco maintains a level of stability in the presence of technology changes.

B. Fixed Supply Characterization

Various forms of fixed battery budget characterization can be directly supported:

- *fixed absolute battery budget*, *e.g.*, **bsupply**(20000) says the execution can consume up to 20000mwh of battery.
- *fixed relative battery budget*, *e.g.*, **bsupply**($0.2*$**battery**) says the execution can consume up to 20% of the remaining battery.
- *maximum battery budget*: *e.g.*, **bsupply**(**battery**) says the execution can as much as the entire remaining battery.

Analogously, simple forms of temperature budget characterization are supported:

- *fixed temperature threshold*, *e.g.*, the use of **tsupply**($70-$**temperature**) says the execution should not increase the CPU temperature to more than 70°C.
- *fixed temperature increase*, *e.g.*, **tsupply**(20) says the execution should not increase the CPU temperature by more than 20°C.
- *fixed temperature increase ratio*, *e.g.*, **tsupply**($1.1*$**temperature**) says the execution should not increase the CPU temperature by more than 10%.

C. Revisable Budget

In Eco, both supply sum and supply gauge are expressions reevaluated every time a calibration is performed. This implies the *budget* (the value of what the supply sum expression evaluates to) does not need to remain constant throughout the execution of the sustainable block. This feature can be useful to adaptively set the budget. For example, a programmer may not know at the beginning what a "fair" budget would be, and she can "test run" a portion of the program to find out and reset the supply budget. Consider the example in Figure 5.

```
double budget = MAX;
old = battery;
sustainable {
  for (int num = 0; num < IMGS; num++) {
    if (num == 0.1*IMGS) {
      budget = (battery − old) * 9;
    }
    ...
  }
}
bsupply (budget)
demand (...) −> (...);
```

Fig. 5. Revisable Budget

```
int workdone = 0;
int workleft = IMGS * res.x * res.y;
sustainable {
  for (int num = 0; num < IMGS; num++) {
    ...
    for (int i = 0; i < IMAX; i++) {
      for (int j = 0; j < JMAX; j++) {
        ...
    }}
    workdone+= IMAX * JMAX;
    workleft = (IMGS−num) * IMAX; JMAX;
    images[num] = image;
  }
}
bsupply (PER * BUDGET)
demand (workdone + workleft) −> (workleft);
```

Fig. 6. Adaptive Demand Characterization

Here, the program decides to executes 10% of the images, and calculate the budget for the remaining 90% of the image processing accordingly.

More broadly, revisable budget programming pattern may also be useful in (1) energy-aware systems relying on renewable energy (*e.g.*, solar), where supply change is not monotone as the sustainable block executes; (2) systems with adaptive cooling devices, where temperature change may also follow complex patterns.

D. Adaptive Demand Characterization

Just like supply, demand sum and demand gauge are also expressions that are reevaluated upon calibration. This allows programmers to adaptively adjust the demand during the sustainable block execution. For instance, Figure 6 shows a refined version of Figure 2, where both the size of the overall demand (*i.e.*, what the demand sum evaluates to) and the size of the remaining demand (*i.e.*, what the demand gauge evaluates to) may be adjusted. Here, we not only consider the number of images as demand indicators, but also the number of pixels in the images. In the new scheme, the change of resolution would alter both the size of the overall demand and the size of the remaining demand, because the adoption of a lower resolution in rendering an image resulted in both the slower-than-expected growth for completed work (`workdone`), and the reduction of the projected work ahead (`workleft`).

E. Non-Linear Behaviors

Battery-powered computer users often observe that the same program execution that reduces battery from 80% to 78%

[1]http://developer.android.com

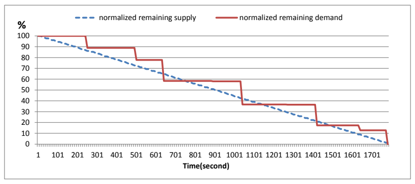

Fig. 7. sunflow (PER = 0.9)

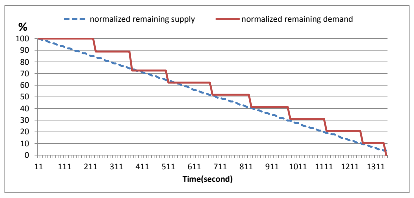

Fig. 8. sunflow (PER = 0.7)

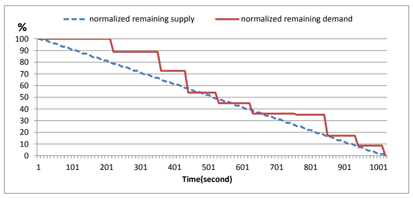

Fig. 9. sunflow (PER = 0.5)

may reduces the same battery from 5% to 1%. Such non-linear battery behaviors are generally minimized by built-in OS/firmware estimators for new batteries, but they may become more pronounced as batteries age. If high precision on battery supply is needed, such variations should not be ignored and can be modeled in Eco as:

```
sustainable {
   double startB = 0.2*getB();
   ...
}
supply (startB) (startB − redress(getB()))
demand (...) (...)
```

where redress is a custom-defined function mapping readings from getBP to the "compensated" values. For example, one simplistic redress function would be to map value b to b if $b \geq 0.2$ and b to $0.9 \times b$ otherwise. As energy-aware programming becomes more prevalent, we envision such "redress" functions would be part of the "battery profile" available to end programmers as library APIs.

A similar programming idiom can be applied for non-linear thermal behaviors. According to solid matter physics, the energy to increase the temperature from 30°C to 35°C and that from 300°C to 305°C significantly differ. (The range of operating temperatures for CPUs is usually much narrower, so the non-linear effect in this area is relatively small.)

The same idea can be applied to demand characterization. Non-linear behaviors can be supported when redress is not a linear function.

VI. Implementation and Evaluation

A. Implementation

Eco is implemented on top of the Polyglot compiler framework 2.5 [28], as an extension to Java. The battery data was queried through Advanced Configuration and Power Interface (ACPI) [1] and read by the program. The CPU temperature was queried with the CoreTemp[2] API, in Celsius. CoreTemp collects data directly from a Digital Thermal Sensor (DTS) located in each individual processing core. There is no need for external measuring devices or external circuit located on the motherboard to report temperature.

B. Benchmarks

We selected the following Java benchmarks and modified them into Eco: (1) sunflow,[3] a rendering system that uses raytracing, (2) jspider,[4] a web crawler, (3) montecarlo,[5] a financial simulation from the Java Grande benchmark suite, (4) xalan,[6] an XSLT processor that converts XML documents, and (5) rasterizer, from the batik[7] distribution, that rasterizes SVG files. Our main selection criterion is diversity, covering domains such as graphics, web, statistics, program transformation, and data transformation. These benchmarks cover a diverse range of applications to support variability in program behaviors, such as image resolution in sunflow, URL request intervals in jspider, sampling size in montecarlo, and rasterization quality in batik. Several selected benchmarks appeared in Dacapo[8] benchmark suite.

All experiments were performed on a Intel 2.53GHz Duo core CPU laptop with 8GB RAM.

C. Battery-Aware Programming

We modified the Java sunflow benchmark with Eco syntax. The rendering logic in sunflow resembles the code snippet in Figure 6. Our benchmarking execution renders 9 images (IMGS=9), and the resolution and depth are implemented as first-class mode cases with identical definitions as in Figure 2.

We first execute the unmodified Java program and use its battery consumption as the baseline supply. This number is used for setting the BUDGET constant in Figure 6. In our experiment, BUDGET = 20970mWh. We execute the Eco program with 90%, 70%, 50% of the BUDGET. It is worth pointing out that the experiment process described here is only meant for inducing "threshold behaviors", *e.g.*, the program running with "just enough" battery, or "slightly inadequate" battery, or "significantly inadequate" battery. An Eco programmer does not need to profile in this manner (or worse, plugging 20970 into the source code). The more intuitive programming idioms were described in Section V.

The time series of supply gauage (blue lines) and demand gauge (red lines) for the sunflow executions with 90%, 70%, 50% of the BUDGET were shown in Figure 7, Figure 8, Figure 9, respectively. All data are normalized. All experiments are started with the same battery level. At any given time, if the demand curve is (significantly) above the supply curve, the program runs in a "deficit." A mode change will happen after the processing of the current task, in the hope that the demand for the next task will go below the supply change. As one can see, battery supply change is near linear. The change in demand happens at discrete time, since workdone and workleft are not updated until an image is processed.

[2]http://www.alcpu.com/CoreTemp
[3]http://sunflow.sourceforge.net/
[4]http://j-spider.sourceforge.net/
[5]http://www.javagrande.org/
[6]http://xalan.apache.org/
[7]http://xmlgraphics.apache.org/batik/
[8]http://www.dacapobench.org/

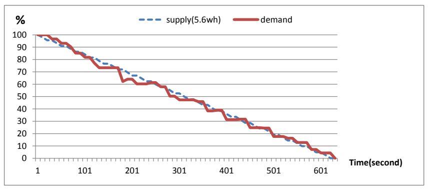

Fig. 10. `batik`: supply/demand (70% budget)

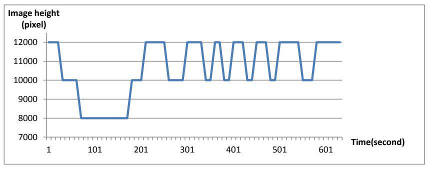

Fig. 11. `batik`: image height (70% budget)

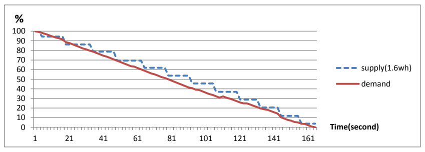

Fig. 12. `montecarlo`: supply/demand (80% budget)

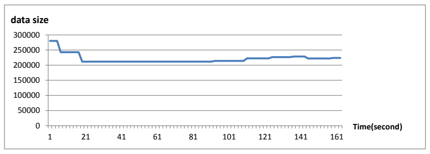

Fig. 13. `montecarlo`: sample size (80% budget)

Therefore, each "plateau" in the demand curve represents the processing of one image. As high-resolution rendering takes significantly longer time, the "width" of each "plateau" is indicative of the operational mode.

Sustainability—the scenario where the supply consistently meets the demand—is maintained in all three scenarios: observe the supply curve and the demand curve in all three figures follow similar patterns of decline. The following table reports the statistics of our experiments:

PER	supply set	supply used	# hi	# mid	# lo
0.9	188.6wh	186.0wh	5	4	0
0.7	147.0wh	142.1wh	1	7	1
0.5	104.8wh	103.3wh	1	2	6

Observe that in all cases, Eco runtime is capable of meeting the budget (*i.e.*, the "supply used" is less than the "supply set"). The `#hi`, `#mid`, `#lo` columns in the table list the number of images rendered with high/mid/low resolutions. Predictably, more images are rendered with lower resolutions as the supply is set lower.

Next, we designed a battery-aware variant of `batik`, processing 27 SVG files with the supply of 8050mwh $\times$ 70%. Similar to the fashion the `sunflow` experiment was constructed, we first executed the unmodified Java program and obtained the battery usage of 8050mwh, which we informally call "the budget." The battery-aware variant thus runs on 70% of that budget. The adaptive behaviors are defined by a mode case value for target image height. Since the image resolution is adjusted by ratio, this value is strongly correlated with the quality of rasterization, as a larger height means a larger width and therefore higher resolution. The mode case is defined by height pixel as $\{$`hi` : 12000; `mid` : 10000; `lo` : 8000$\}$, where 12000 is used for the original Java execution. As Figure 10 suggests, Eco achieves the goal of sustainability by matching supply and demand closely. Figure 11 shows the adaptively selected resolutions based on the changes of supply and demand.

Last, we modified `montecarlo`, where a mode case is defined to adjust the sample size. Specifically, the value in each case member is the difference between the "perfect" sample size (280,000) and the sample size being used for that mode. The larger sample size, the higher quality the output will be. We use a syntactic sugar to define this mode case (with a large number of cases whose mode names are implicit), in the same effect of defining $\{\mathtt{m}_0 : 0; \ldots \mathtt{m}_i : i \times \Delta; \ldots \mathtt{m}_n : n \times \Delta\}$ where $\Delta = 10,000$. In other words, our benchmark adjusts the sample size by the increment/decrement of Δ. The execution of the unmodified Java code (for Monte Carlo core algorithm) consumes 2060mwh of battery ("the budget"). The Eco execution operates on of 80% of that budget, with results shown in Figure 12 and Figure 13. As Figure 12 suggests, Eco again matches supply and demand closely. Figure 13 shows the change of sample size over time.

D. Temperature-Aware Programming

To demonstrate temperature-aware programming, we use `xalan` to transform 17 XML files with 8000 iterations. We modify the original Java program as follows. First, we set a fixed temperature threshold to 60°C. This is expressed as **tsupply**(60 − **temperature**). Second, in between every other file transformation, we allow the CPU to sleep at a fixed interval. A mode case is used for this sleep interval, set with two case members: when the mode is `hi`, the interval is set at 4 milliseconds, whereas when the mode is `lo`, the interval is set at 10 milliseconds.

Figure 14 demonstrates the temperature changes of `xalan`, both in the Java execution and the Eco execution. As a benchmark, `xalan` involves intensive CPU-bound computations. Without any sustainability management, the Java execution drives the temperature over 60°C within one minute (a rise of about 15°C). The Eco execution on the other hand successfully maintains the CPU temeperature within the threshold. Figure 15 may explain the fundamental difference between the two executions, showing different levels of CPU utilization (Y-axis, plotted with OS performance monitor). During the Java execution, the CPU utilization is mostly 100%. The Eco execution however regulates the CPU utilization at a much lower adaptive level.

The figures here also demonstrate the trade-off between temperature regulation and performance. Without any sleep, the Java execution is able to complete the transformation within about half the time of the Eco execution (terminates after around 301 seconds). This comes with no surprise: the latter execution intentionally slows down itself frequently. Figure 16 demonstrates the change of sleep intervals over time.

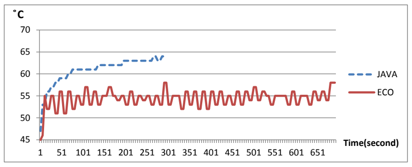

Fig. 14. `xalan`: CPU temperature

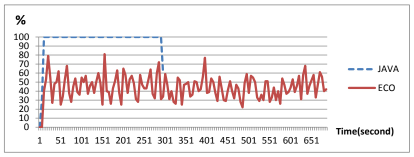

Fig. 15. `xalan`: CPU utilization

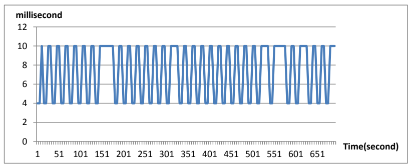

Fig. 16. `xalan`: sleep interval

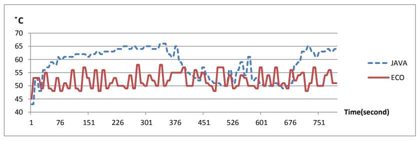

Fig. 17. `jspider`: temperature

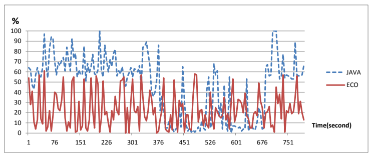

Fig. 18. `jspider`: CPU utilization

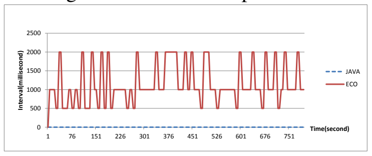

Fig. 19. `jspider`: sleep interval

We also constructed a temperature-aware execution of `jspider` in Eco. Similar to `xalan`, the temperature is also set at the fixed threshold of 60°C. We employ a similar strategy for temperature control, by allowing the crawler to sleep at fixed intervals. Unlike `xalan` where each XML file is processed in a relatively short period of time, each step in crawling—involving requesting a URL, parsing the webpage, and analyzing the links contained in the page—takes significantly longer. This allows us to design with longer intervals of sleeps (but much less frequent than `xalan`). The interval is implemented as a mode case, with `hi` mapping to 0.5 second, `mid` to 1 second, and `hi` to 2 seconds. Figure 17 shows the temperature results. Figure 18 shows the CPU utilization, and Figure 19 shows the interval change. Typical for a web crawler, there is no "completion" time unless the user stops the execution. Our demand is set (and reset) at fixed time intervals. As a result, all time series shown here for `jspider` can be viewed as the first 800 seconds of an (almost infinitely running) execution.

E. Programming Efforts

The programming effort for Eco is mild, and the LOCs that involve Eco-specific changes are relatively small:

LOCs	`sunflow`	`jspider`	`montecarlo`	`xalan`	`batik`
original	30984	13986	3128	347801	325238
Eco changes	46	56	23	32	27

Despite some benchmarks have high LOCs in their Java versions, the changes involved for sustainable programming concentrate on a very small number of files, typically the entry points of core algorithms, and the variables representing the adjustable parameters of those algorithms. In our experience, the majority of time was used in understanding existing benchmarks, and finding appropriate procedures/algorithms to support adaptive sustainable programming. Once the algorithm of interest is identified, the programming effort itself for developing the benchmarks we experimented with — adding Eco-specific expressions into programs, and debugging — is small, usually in hours.

VII. Related Work

Sustainable computing is an emerging topic in computer science. Within software engineering, support for sustainability has appeared in requirements engineering [34], [29], [32] and software architecture [18], [20]. At lower levels of the compute stack, modeling computer systems with supply and demand is mostly known in operating systems. Examples include resource management and scheduling for grid environment [6], device energy management [40], utility computing in clouds [7], energy distribution among users [27]. Recently, the supply/demand model is also used for performance tuning [26]. Eco enriches the landscape of sustainable computing by empowering *programmers*.

A number of programming models exist for developing energy-aware software, such as Green [4], EnerJ [36], and ET [10]. They do not offer fine-grained energy/temperature budgeting to programmers, and do not focus on maintaining sustainability through programmer supply/demand characterization. Beyond energy-aware computing, programming models for self-adaptive systems (*e.g.*, [8]) share a high-level design goal of Eco: promoting adaptiveness in the presence of dynamic variations of software/system environments.

Broadly, application-level energy management and optimization is an active area, with recent results diversely ranging from program analysis and optimization (*e.g.*, [16], [5]), testing and debugging (*e.g.*, [21], [2]), design patterns (*e.g.*, [35], [24]), decision framework (*e.g.*, [25]), measurement and estimation support (*e.g.*, [37], [17]), runtime support (*e.g.*, [33]), and empirical studies (*e.g.*, [30], [31], [23]).

More broadly, there is a long history in VLSI, architecture, and OS research to design and implement battery-aware (*e.g.* [12]) or temperature-aware (*e.g.* [38]) computer systems. A fundamental challenge is the need to "predict the future," with solutions ranging from profiling and monitoring [11], machine learning [9], to agent-based modeling [13]. Eco complements existing systems by bringing programmer knowledge into the design space.

VIII. Conclusion

This paper describes Eco, a sustainable programming model for developing Java-like applications. With novel abstractions for supply and demand shaping, Eco brings programmers into the loop of sustainability management and promotes energy-aware programming and temperature-aware programming.

Acknowledgments We thank the anonymous reviewers for their useful comments, and Anthony Canino for his help improving the compiler and the benchmarks. This work is partially supported by US NSF CCF-1054515.

References

[1] Advanced configuration and power interface, http://www.acpi.info.

[2] Carat, http://carat.cs.berkeley.edu.

[3] Data centers look for lower-emission cooling, http://www.nytimes.com/2011/06/20/business/global/20green.html.

[4] Baek, W., and Chilimbi, T. M. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI'10* (2010), pp. 198–209.

[5] Bartenstein, T., and Liu, Y. D. Green streams for data-intensive software. In *ICSE'13* (May 2013).

[6] Buyya, R., Abramson, D., Giddy, J., and Stockinger, H. Economic models for resource management and scheduling in grid computing. *Concurrency and computation: practice and experience 14*, 13-15 (2002), 1507–1542.

[7] Buyya, R., Yeo, C. S., and Venugopal, S. Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. In *Proceedings of the 2008 10th IEEE International Conference on High Performance Computing and Communications* (2008), HPCC '08, pp. 5–13.

[8] Cheng, S.-W., and Garlan, D. Stitch: A language for architecture-based self-adaptation. *J. Syst. Softw. 85*, 12 (Dec. 2012), 2860–2875.

[9] Chung, E.-Y., Benini, L., and De Micheli, G. Dynamic power management using adaptive learning tree. In *Proceedings of the 1999 IEEE/ACM international conference on Computer-aided design* (1999), ICCAD '99, pp. 274–279.

[10] Cohen, M., Zhu, H. S., Emgin, S. E., and Liu, Y. D. Energy types. In *OOPSLA '12* (October 2012).

[11] Contreras, G., and Martonosi, M. Power prediction for Intel XScale processors using performance monitoring unit events. In *Proceedings of the 2005 international symposium on Low power electronics and design* (2005), ISLPED '05, pp. 221–226.

[12] Flinn, J., and Satyanarayanan, M. Managing battery lifetime with energy-aware adaptation. *ACM Trans. Comput. Syst. 22*, 2 (May 2004), 137–179.

[13] Ge, Y., Qiu, Q., and Wu, Q. A multi-agent framework for thermal aware task migration in many-core systems. *IEEE Trans. Very Large Scale Integr. Syst. 20*, 10 (Oct. 2012), 1758–1771.

[14] Glover, J. D. D., and Sarma, M. S. *Power System Analysis and Design*, 3rd ed. Brooks/Cole Publishing Co., Pacific Grove, CA, USA, 2001.

[15] Gray, J. The transaction concept: virtues and limitations (invited paper). In *Proceedings of the seventh international conference on Very Large Data Bases - Volume 7* (1981), VLDB '81, pp. 144–154.

[16] Hao, S., Li, D., Halfond, W. G. J., and Govindan, R. Estimating mobile application energy consumption using program analysis. In *ICSE '13* (2013), pp. 92–101.

[17] Hindle, A., Wilson, A., Rasmussen, K., Barlow, E. J., Campbell, J. C., and Romansky, S. Greenminer: A hardware based mining software repositories software energy consumption framework. In *MSR 2014* (2014), pp. 12–21.

[18] Koziolek, H. Sustainability evaluation of software architectures: A systematic review. In *Proceedings of the Joint ACM SIGSOFT Conference – QoSA and ACM SIGSOFT Symposium – ISARCS on Quality of Software Architectures – QoSA and Architecting Critical Systems – ISARCS* (2011), QoSA-ISARCS '11, pp. 3–12.

[19] Kulkarni, A., Liu, Y. D., and Smith, S. F. Task types for pervasive atomicity. In *OOPSLA'10* (October 2010).

[20] Lago, P., Jansen, T., and Jansen, M. The service greenery-integrating sustainability in service oriented software. In *International Workshop on Software Research and Climate Change (WSRCC), co-located with ICSE* (2010), vol. 2.

[21] Li, D., Jin, Y., Sahin, C., Clause, J., and Halfond, W. G. J. Integrated energy-directed test suite optimization. In *ISSTA'14* (2014), pp. 339–350.

[22] Liskov, B. Distributed programming in argus. *Commun. ACM 31*, 3 (1988), 300–312.

[23] Liu, K., Pinto, G., and Liu, Y. D. Data-oriented characterization of application-level energy optimization. In *FASE 2015* (Apr. 2015).

[24] Liu, Y. D. Energy-efficient synchronization through program patterns. In *First International Workshop on Green and Sustainable Software, (GREENS 2012)* (2012), pp. 35–40.

[25] Manotas, I., Pollock, L., and Clause, J. Seeds: A software engineer's energy-optimization decision support framework. In *ICSE'14* (2014), pp. 503–514.

[26] Mitchell, N., and Sweeney, P. F. On the fly capacity planning. In *OOPSLA '13* (October 2013).

[27] Narasimhan, S., McIntyre, D. R., Wolff, F. G., Zhou, Y., Weyer, D. J., and Bhunia, S. A supply-demand model based scalable energy management system for improved energy utilization efficiency. In *International Conference on Green Computing* (2010), IEEE, pp. 97–105.

[28] Nystrom, N., Clarkson, M. R., and Myers, A. C. Polyglot: An extensible compiler framework for java. In *International Conference on Compiler Construction* (Apr. 2003), pp. 138–152.

[29] Penzenstadler, B., and Femmer, H. A generic model for sustainability with process- and product-specific instances. In *Proceedings of the 2013 Workshop on Green in/by Software Engineering* (2013), GIBSE '13, pp. 3–8.

[30] Pinto, G., Castor, F., and Liu, Y. D. Mining questions about software energy consumption. In *MSR'14* (2014), pp. 22–31.

[31] Pinto, G., Castor, F., and Liu, Y. D. Understanding energy behaviors of thread management constructs. In *OOPSLA'14* (2014).

[32] Raturi, A., Penzenstadler, B., Tomlinson, B., and Richardson, D. Developing a sustainability non-functional requirements framework. In *GREENS 2014* (2014), pp. 1–8.

[33] Ribic, H., and Liu, Y. D. Energy-efficient work-stealing language runtimes. In *ASPLOS* (2014), pp. 513–528.

[34] Roher, K., and Richardson, D. Sustainability requirement patterns. In *Requirements Patterns (RePa), 2013 IEEE Third International Workshop on* (July 2013), pp. 8–11.

[35] Sahin, C., Cayci, F., Gutiérrez, I. L. M., Clause, J., Kiamilev, F. E., Pollock, L. L., and Winbladh, K. Initial explorations on design pattern energy usage. In *First International Workshop on Green and Sustainable Software, (GREENS 2012)* (2012), pp. 55–61.

[36] Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., and Grossman, D. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI'11* (June 2011).

[37] Seo, C., Malek, S., and Medvidovic, N. An energy consumption framework for distributed java-based systems. In *ASE '07* (2007), pp. 421–424.

[38] Skadron, K., Stan, M. R., Huang, W., Velusamy, S., Sankaranarayanan, K., and Tarjan, D. Temperature-aware microarchitecture. In *ISCA '03* (2003), pp. 2–13.

[39] Sorber, J., Kostadinov, A., Garber, M., Brennan, M., Corner, M. D., and Berger, E. D. Eon: a language and runtime system for perpetual systems. In *SenSys*, pp. 161–174.

[40] Zeng, H., Ellis, C. S., Lebeck, A. R., and Vahdat, A. Ecosystem: managing energy as a first class operating system resource. In *ASPLOS* (2002), pp. 123–132.

[41] Zhu, H. S., Lin, C., and Liu, Y. D. A programming model for sustainable software (technical report), available online at `https://github.com/pl-eco/ECO`, 2015.

Appendix

To see why the second support is important, consider the example in Figure 20. Here, the characterizer of the sustainable block lexically belongs to class `X`, whereas the calibration expression appears in class `Y`. It is important that during the execution of the method `n` invocation in an `Y` object, the sustainability management as defined by the characterizer in `X` remains in place. Similarly, if we replace the sustainable block in the example with a uniform block, it is important that during the execution of the method `m`, uniformity is still enforced. The S-stack design flexibly enables sustainability management for all program points reachable from a sustainable/uniform block.

A. Abstract Syntax

The core abstract syntax of Eco is defined in Figure 21. A sustainable block in the formal system is represented by **sustainable** $\{e\}(c)$, where e is the body of the sustainable

```
class X {
   void m() {
      Z z = new Z();
        sustainable {
           Y y = new Y();
           y.n(z);
        }
        bsupply (0.2 * battery)
        demand (z.sum) -> (z.gauge);
   }
}
class Y {
   void n(Z z1) {
      ...
      calib (z1.gauge--);
   }
}
class Z {
   int sum;
   int gauge;
      ...
}
```

Fig. 20. Sustainability Management Across Lexical Scopes

e	$::=$	**sustainable** $\{e\}(c)$	*expression*
	$\|$	**uniform**$\{e\}$	
	$\|$	$\{\overline{\mathtt{m} : e}\}^{\tau}$	
	$\|$	**select** e	
	$\|$	**calib** e	
	$\|$	$\mathtt{x} \mid e.\mathtt{fd} \mid e.\mathtt{md}(e) \mid$ **new** $\mathtt{X}$	
	$\|$	$n \mid\ e.\mathtt{fd} = e \mid \mathtt{x} = e \mid e; e$	
c	$::=$	$\langle e; e; e; e\rangle$	*characterizer*
P	$::=$	$\langle \overline{C}; \overline{D}; e\rangle$	*program*
C	$::=$	**class** $\mathtt{X}$ **extends** $\mathtt{Y}$ $\{\overline{F}\ \overline{M}\}$	*class*
D	$::=$	**mode** $\mathtt{m} <: \mathtt{m}$	*mode declaration*
F	$::=$	$\tau\ \mathtt{fd}$	*fields*
M	$::=$	$\tau\ \mathtt{md}(\tau\ \mathtt{x})\{\overline{L}\ e\}$	*methods*
τ	$::=$	**num** $\mid \mathtt{X} \mid$ **mcase**$\langle\tau\rangle$	*type*
L	$::=$	$\tau\ \mathtt{x}$	*local declaration*
$\mathtt{X}, \mathtt{Y}$	$\in$	$\mathbb{CN} \cup \{\mathtt{Object}\}$	*class name*
$\mathtt{x}, \mathtt{y}$	$\in$	$\mathbb{VN} \cup \{$**this**$\}$	*variable name*
$\mathtt{fd}$	$\in$	$\mathbb{FN}$	*field name*
$\mathtt{md}$	$\in$	$\mathbb{MN}$	*method name*
$\mathtt{m}$	$\in$	$\mathbb{MDN}$	*mode name*
n	$\in$	$\mathbb{NUM}$	*numeric constant*

Fig. 21. Eco Core Abstract Syntax

block, and c is the *characterizer*. The latter is a 4-tuple $\langle e_1; e_2; e_3; e_4\rangle$ where e_1, e_3, e_4 are the expressions for representing the *supply sum*, *demand sum*, *demand gauge* we informally introduced in the previous section, respectively. The additional expression e_2 is called the *supply gauge*, an expression to indicate the remaining supply. The 4-tuple demand/supply characterization is slightly more general than the concrete syntax we used for the sustainable block in the Figure 2 example.

Two other Eco expressions are used in the previous example. Expression **uniform**$\{e\}$ defines a uniform block whose body is e. Expression $\{\overline{\mathtt{m} : e}\}^{\tau}$ represents a mode case, where each element $\mathtt{m} : e$ in the sequence is called a *case member*. For simplicity, we associate the mode case with type τ, the (common) type of every expression in the case members.

To facilitate the formal presentation, we introduce two additional syntactical forms. Expression **select** e destructs a mode case represented by e, and expression **calib** e is a "checkpoint," *i.e.*, performing a sustainability check after evaluating e. Our formal system takes a two-step approach to present language semantics. We first present an operational semantics (in Section B) that programmers need to manually destruct mode cases (through **select** e) and manually perform calibration (through **calib** e).

An Eco program consists of a sequence of classes ($\overline{C}$), a sequence of mode declarations ($\overline{D}$), and a bootstrapping expression e. The "overline" notation $\overline{a}$ represents a sequence of elements represented by metavariable a. We require the modes defined in $\overline{D}$ form a total order, captured by predicate $\textsc{total}(\overline{D})$. Formally, the predicate is true iff the reflexive and transitive closure of the smallest relation containing elements $\langle \mathtt{m}; \mathtt{m}'\rangle$ where **mode** $\mathtt{m} <: \mathtt{m}' \in \overline{D}$ is a total order. We further call the least element and greatest element in the total order the *least mode* and *greatest mode* respectively. Implicitly parameterized by $\overline{D}$, unary operators $\uparrow$ and $\downarrow$ are standard successive and precedent operators over total order. Furthermore, $\uparrow \mathtt{m} \overset{\text{def}}{=} \mathtt{m}$ if $\mathtt{m}$ is the greatest mode and $\downarrow \mathtt{m} \overset{\text{def}}{=} \mathtt{m}$ if $\mathtt{m}$ is the least mode.

Type τ is either primitive **num**, or an object type $\mathtt{X}$, or a type for the mode case value **mcase**$\langle\tau\rangle$. The last type form says that the mode case has case members whose types are τ.

a) Standard Features and Notations: The rest of the Eco syntax is standard, similar to the choices of Featherweight Java (FJ).

Additional expressions are variables $\mathtt{x}$ (subsuming special name **this** for self reference), field read expression $e.\mathtt{fd}$, method invocation expression $e.\mathtt{md}(e)$, and instantiation expression **new** $\mathtt{X}$. We omit verbose and orthogonal features such as constructors and multi-argument methods. To elucidate ideas relevant to Eco, we explicitly support numeric constant n, assignment expression $\mathtt{x} = e$, field write $e.\mathtt{fd} = e$, and continuation $e; e$.

The definition for a class is nearly identical to FJ, with pre-defined class name `Object` as the root class. Our only addition is to make local variable declarations in a method body explicit, where a local variable declaration is represented by metavariable L. Throughout the paper we liberally use three FJ functions: (1) $\textsc{fields}(\mathtt{X})$ computes the field declarations for class $\mathtt{X}$; (2) $\textsc{mtype}(\mathtt{md}, \mathtt{X})$ computes the signature for method named $\mathtt{md}$ of class named $\mathtt{X}$, and the computed signature takes the form of $\tau \to \tau'$, where τ and τ' are the argument/return type respectively; (3) $\textsc{mbody}(\mathtt{md}, \mathtt{X})$ computes the method body for method named $\mathtt{md}$ of class named $\mathtt{X}$. The computed body takes the form of $\mathtt{x}.\overline{L}.e$, where $\mathtt{x}$ is the name of the formal argument, $\overline{L}$ is the local variable declarations, and e is the expression that constitutes the method body. The definitions of all three functions are identical to FJ's counterparts, except that what MBODY computes in our case also includes the local declarations associated with the method body.

We use notation $[a_1, \ldots, a_n]$ to represent a sequence of elements a_1, …, a_n, and use comma (,) for sequence concatenation. When no confusion can arise, we use set notations $\in$, $\subseteq$ over sequences as well. Given sequence $Q = [a_1 \mapsto b_1, \ldots a_n \mapsto b_n]$, we define $\textsc{dom}(Q) \overset{\text{def}}{=} \{a_1, \ldots, a_n\}$. Further we define $Q[a_i \mapsto b'_i] \overset{\text{def}}{=} [a_1 \mapsto b_1, \ldots, a_{i-1} \mapsto b_{i-1}, a_i \mapsto b'_i, a_{i+1} \mapsto b_{i+1}, \ldots, a_n \mapsto b_n]$ for $1 \leq i \leq n$ and the function is undefined otherwise. Given two sequences Q_1 and Q_2, $Q_1 \uplus Q_2 \overset{\text{def}}{=} Q_1, Q_2$ if $\textsc{dom}(Q_1) \cap \textsc{dom}(Q_2) = \emptyset$; it is undefined otherwise.

B. Operational Semantics

$$
\begin{array}{lcll}
\Psi & ::= & \langle H; A; S; e\rangle & \textit{runtime configuration} \\
H & ::= & \overline{o \mapsto \langle \mathtt{X}; \sigma\rangle} & \textit{heap} \\
A & ::= & \overline{\mathtt{x} \mapsto v} & \textit{activation record} \\
S & ::= & \overline{s} & \textit{S stack} \\
s & ::= & e : \mathtt{m} \mid \perp & \textit{S-stack entry} \\
\sigma & ::= & \overline{\mathtt{fd} \mapsto v} & \textit{field store} \\
v & ::= & n \mid o \mid \textbf{null} \mid \{\overline{\mathtt{m} : v}\}^{\tau} & \textit{values} \\
 & \mid & \langle v; v; v; v\rangle & \\
e & ::= & \cdots \mid v \mid e?e \mid c & \textit{extended expressions} \\
 & \mid & e\lightning\mathtt{m} \mid \textbf{in}(A, e) & \\
\mathbf{E} & ::= & \circ & \textit{evaluation context} \\
 & \mid & \{\overline{\mathtt{m} : v}, \mathtt{m} : \mathbf{E}, \overline{\mathtt{m} : e}\}^{\tau} & \\
 & \mid & \mathbf{E}.\mathtt{fd} & \\
 & \mid & \mathbf{E}.\mathtt{fd} = e \mid v.\mathtt{fd} = \mathbf{E} & \\
 & \mid & \mathbf{E}.\mathtt{md}(e) \mid v.\mathtt{md}(\mathbf{E}) & \\
 & \mid & \mathtt{x} = \mathbf{E} \mid \mathbf{E}\lightning \mid \mathbf{E}; e & \\
 & \mid & \textbf{select } \mathbf{E} & \\
 & \mid & \textbf{calib } \mathbf{E} & \\
 & \mid & \langle v; \ldots \mathbf{E}; \ldots; e\rangle & \\
 & \mid & e?\mathbf{E} \mid \mathbf{E}?v &
\end{array}
$$

$\boxed{\text{Heap Access Operators (Read and Write)}}$

$$
\begin{array}{rcl}
H\{o, \mathtt{fd}\} & \overset{\text{def}}{=} & \sigma(\mathtt{fd}) \text{ where } H(o) = \langle \mathtt{X}; \sigma\rangle \\
H\{o, \mathtt{fd} \to v\} & \overset{\text{def}}{=} & H[o \mapsto \langle \mathtt{X}; \sigma[\mathtt{fd} \mapsto v]\rangle] \\
 & \text{if} & H(o) = \langle \mathtt{X}; \sigma\rangle \\
H + \{o, \mathtt{X}\} & \overset{\text{def}}{=} & H \uplus (o \mapsto \langle \mathtt{X}; \biguplus_{\tau\ \mathtt{fd} \in \textsc{dom}(\textsc{fields}(\mathtt{X}))} \overline{\mathtt{fd} \mapsto \textbf{null}}\rangle)
\end{array}
$$

Fig. 22. Eco Runtime Definitions

Figure 22 defines the data structures relevant to the Eco runtime. A runtime configuration consists of a *heap H*, an *activation record A*, a *sustainability stack S*, and an expression e. The heap maps each object ID to its class name, together with a field store that maps field names to values. An activation record maps local variable names to values. The sustainability stack — or S-stack for short — is a key data structure to maintain the contexts of sustainable/uniform blocks. Intuitively, each entry of this stack models the execution state of a sustainable block. Structurally, each entry consists of two elements: the characterizer expression of the sustainable block and the mode of the current sustainable block execution, called the *operational mode*. We will explain the details behind this data structure shortly.

Values can either be a number n, an object ID o, **null** value, a mode case where every case member is a value. For convenience of defining operating semantics, we extend expressions to include values, and several additional auxiliary expressions. Expression $e?e'$ says the operational mode of a sustainable block should be determined by e, and the result of the expression should be the value computed by evaluating e'. Expression $e\lightning$ says that the S-stack should be popped after the evaluation of e. Expression $\textbf{in}(A, e)$ says evaluating expression e under the activation record of A. As another convenience of formalism, we liberally treat characterizer (a 4-tuple) as an expression, and a value if all tuple components are values.

Common heap access operators are defined in the same Figure. Heap operator $H\{o, \mathtt{fd}\}$ computes the value of field $\mathtt{fd}$ of object o in heap H, and $H\{o, \mathtt{fd} \to v\}$ computes an updated heap where the value of $\mathtt{fd}$ is updated to v, whereas $H + \{o, \mathtt{X}\}$ adds an additional entry to H, mapping o to class name $\mathtt{X}$ and an initialized field store.

Figure 23 defines the operational semantics of Eco, where notation $\Psi \rightsquigarrow \Psi'$ means runtime configuration Ψ one-step reduces to Ψ'. All rules are implicitly parameterized by $\overline{C}$ and $\overline{D}$.

b) Evaluation Context: Reduction rule (R-Cxt) defines reduction over an evaluation context. Evaluation context $\mathbf{E}$ is defined in Figure 22, as either a hole $\circ$, or an expression with a hole inside. Notation $\mathbf{E}[e]$ means replacing the whole in $\mathbf{E}$ with expression e.

Evaluation contexts are known to be useful in defining evaluation order. For instance, according to the case of $\mathbf{E}$ immediately after the hole, all expressions in case members are evaluated at mode case definition time. As another example, the last two cases of $\mathbf{E}$ says expression $e_1?e_2$ will be evaluated by evaluating e_2 first, then e_1, then the expression itself.

c) Sustainable and Uniform Blocks: Upon the entry of a sustainable block, Eco pushes one item onto the sustainability stack as demonstrated by (R-Sustain). The new stack entry contains the characterizer information of the sustainable block, which will be used when **calib** expressions are encountered during the sustainable block execution. This design flexibly supports sustainability management when the sustainable block and the calibration expression do not belong to the same lexical scope.

The initial operational mode of the block is the same as the "current" operational mode before the block is entered, where the notion of "current" is defined by the following function:

$$
\begin{array}{lcll}
\textsc{mnow}(S) & \overset{\text{def}}{=} & \mathtt{m} & \text{if } S = S', (e : \mathtt{m}) \\
\textsc{mnow}(S) & \overset{\text{def}}{=} & \textsc{mnow}(S') & \text{if } S = S', \perp
\end{array}
$$

(R-Cxt)	$H, A, S, \mathbf{E}[e]$	$\leadsto$	$H', A', S', \mathbf{E}[e']$	if $H, A, S, e \leadsto H', A', S', e'$
(R-Sustain)	$H, A, S, \textbf{sustainable } \{e\}(c)$	$\leadsto$	$H, A, (S, \textbf{in}(A, c) : \mathtt{m}), e\lightning$	if $\textsc{mnow}(S) = \mathtt{m}$
(R-Uniform)	$H, A, S, \textbf{uniform } \{e\}$	$\leadsto$	$H, A, (S, \perp), e\lightning$	
(R-Calib)	$H, A, S, \textbf{calib } v$	$\leadsto$	$H, A, S, e_1? \dots (e_n?v)$	$S = [e_1 : \mathtt{m}_1, \dots e_n : \mathtt{m}_n]$
(R-NoCalib)	$H, A, S, \textbf{calib } v$	$\leadsto$	H, A, S, v	if $\perp \in S$
(R-Adjust1)	$H, A, S, c?v$	$\leadsto$	H, A, S, v	if $\textsc{match}(c)$
(R-Adjust2)	$H, A, (S, e : \mathtt{m}), c?v$	$\leadsto$	$H, A, (S, \downarrow e : \mathtt{m}), v$	if $\textsc{deficit}(c)$
(R-Adjust3)	$H, A, (S, e : \mathtt{m}), c?v$	$\leadsto$	$H, A, (S, \uparrow e : \mathtt{m}), v$	if $\textsc{surplus}(c)$
(R-MSelect)	$H, A, S, \textbf{select } v$	$\leadsto$	$H, A, S, v/\textsc{mnow}(S)$	
(R-FRead)	$H, A, S, o.\mathtt{fd}$	$\leadsto$	$H, A, S, H\{o, \mathtt{fd}\}$	
(R-FWrite)	$H, A, S, o.\mathtt{fd} = v$	$\leadsto$	$H\{o, \mathtt{fd} \to v\}, A, S, v$	
(R-VRead)	$H, A, S, \mathtt{x}$	$\leadsto$	$H, A, S, A(\mathtt{x})$	
(R-VWrite)	$H, A, S, \mathtt{x} = v$	$\leadsto$	$H, A[\mathtt{x} \to v], S, v$	
(R-New)	$H, A, S, \textbf{new } \mathtt{X}$	$\leadsto$	$H + \{o, \mathtt{X}\}, A, S, o$	if o fresh
(R-Msg)	$H, A, S, o.\mathtt{md}(v)$	$\leadsto$	$H, A, S, \textbf{in}(A', e)$	if $H(o) = \langle \mathtt{X}; \sigma \rangle, \textsc{mbody}(\mathtt{md}, \mathtt{X}) = \mathtt{x}.\overline{\tau\ \mathtt{y}}.e$ $A' = [\textbf{this} \mapsto o, \mathtt{x} \mapsto v] \uplus \overline{\mathtt{y} \mapsto \textbf{null}}$
(R-InE)	$H, A, S, \textbf{in}(A_0, e)$	$\leadsto$	$H', A, S', \textbf{in}(A'_0, e')$	if $H, A_0, S, e \leadsto H', A'_0, S', e'$
(R-InV)	$H, A, S, \textbf{in}(A_0, v)$	$\leadsto$	H, A, S, v	
(R-MPop)	$H, A, (S, s), v\lightning$	$\leadsto$	H, A, S, v	
(R-Cont)	$H, A, S, v; e$	$\leadsto$	H, A, S, e	

Fig. 23. Eco Operational Semantics

This design is useful in the presence of nested sustainable blocks, where the inner block starts operating at a mode sustainable for the outer block. When the program bootstraps, we assign the greatest mode in the mode total order as the operational mode, with details in Section E.

Whenever a uniform block is encountered, the S-stack is pushed with $\perp$, a special S-stack entry. This is demonstrated in (R-Uniform).

d) Supply/Demand Matching: The matching between supply and demand is captured by the following predicates, where numeric value n_1 corresponds to supply sum, n'_1 to supply gauge, n_2 to demand sum, and n'_2 to demand gauge, and ϵ is a small positive numeric constant close to 0:

$$\begin{aligned}
\textsc{match}(\langle n_1; n'_1; n_2; n'_2 \rangle) &\overset{\text{def}}{=} \left|\frac{n'_1}{n_1} - \frac{n'_2}{n_2}\right| \le \epsilon \\
\textsc{surplus}(\langle n_1; n'_1; n_2; n'_2 \rangle) &\overset{\text{def}}{=} \frac{n'_1}{n_1} - \frac{n'_2}{n_2} > \epsilon \\
\textsc{deficit}(\langle n_1; n'_1; n_2; n'_2 \rangle) &\overset{\text{def}}{=} \frac{n'_2}{n_2} - \frac{n'_1}{n_1} > \epsilon
\end{aligned}$$

Intuitively, the supply and the demand match when the proportion of remaining supply ($\frac{n'_1}{n_1}$) and the proportion of remaining demand ($\frac{n'_2}{n_2}$) are close in range. If the former significantly exceeds the latter, we have a "surplus" of supply. If the latter significantly exceeds the former, we have a "deficit" of supply. In real number domains, equality is established through approximity in distance. In practice, a small yet not minuscule ϵ can help increase the stability of the system—otherwise every supply/demand comparison would yield either a surplus or deficit. Eco language implementation selects $\epsilon = 0.1$: the program is considered "sustainable" if the remaining proportions of supply and demand are within $\pm 10\%$ difference. The calibration will be performed in one of the three cases, defined by (R-Adjust1), (R-Adjust2), and (R-Adjust3).

The language is neutral on the concrete "unit" of the supply or demand. For instance, a supply can either be battery capacity in joules, battery capacity in milliwatt hour, or battery remaining time in seconds, or temperature in celsius, *etc*. The only requirement to keep the definitions above sensible is the supply sum and supply gauge refer to the same unit. Similarly, the programmer is also at the liberty to decide on the "unit" of demand, as long as the demand sum and the demand gauge refer to the same unit. We will come back to this topic in Section V-A.

Calibration is triggered by two reduction rules: (R-Calib) and (R-NoCalib). In the first rule, all characterizers on the S-stack will be evaluated for calibration, with the top item on S-stack being evaluated first. The definition here subsumes the case where sustainable blocks are nested (either lexically or encountered dynamically), so that sustainability management applies to all blocks, with the most immediate (inner) block first. Rule (R-NoCalib) says no calibration is needed if the execution is part of the **uniform** block execution.

e) Mode Case Destruction: Mode cases are destructed based on the operational mode, as defined in (R-Select). The simple operator appearing in that rule is defined as:

$$\{\mathtt{m}_1 : v_1, \dots, \mathtt{m}_n : v_n\}/\mathtt{m} \overset{\text{def}}{=} v_i$$

where $\mathtt{m}_i$ is the least mode among $\{\mathtt{m}_1, \dots, \mathtt{m}_n\}$ that is equal to or greater than $\mathtt{m}$ according to the total order defined by the program.

This definition captures two intuitive facts: (i) if the mode case explicitly includes a case member whose label is the operational mode, that case should be selected; (ii) otherwise, the case member whose mode is greater to the operational mode—but closest to it—should be selected.

Recall that the program bootstraps with the greatest mode. This implies when a **select** expression appears outside of any sustainable block, the greatest mode is used. This aligns with our intuition that code not subject to sustainability management should not be approximated.

f) Other Rules: The other rules are standard. (R-FRead) and (R-FWrite) define field read/write, whereas (R-VRead)

$$
\begin{array}{rcl}
[\![\,\textbf{sustainable}\ \{e\}(c)\,]\!]_\delta & = & \textbf{sustainable}\ \{[\![\,e\,]\!]_{\delta\cup \mathrm{FV}(e_4)}\}(c) \\
 & & \text{if}\ \ c = \langle e_1; e_2; e_3; e_4\rangle \\
[\![\,\textbf{uniform}\ \{e\}\,]\!]_\delta & = & \textbf{uniform}\ \{e\} \\
[\![\,\textbf{select}\ e\,]\!]_\delta & = & \textbf{select}\ [\![\,e\,]\!]_\delta \\
[\![\,\mathtt{x}\,]\!]_\delta & = & \mathtt{x} \\
[\![\,\textbf{new}\ \mathtt{X}\,]\!]_\delta & = & \textbf{new}\ \mathtt{X} \\
[\![\,e.\mathtt{md}(e')\,]\!]_\delta & = & [\![\,e\,]\!]_\delta.\mathtt{md}([\![\,e'\,]\!]_\delta) \\
[\![\,e.\mathtt{fd}\,]\!]_\delta & = & [\![\,e\,]\!]_\delta.\mathtt{fd} \\
[\![\,e; e'\,]\!]_\delta & = & [\![\,e\,]\!]_\delta; [\![\,e'\,]\!]_\delta \\
[\![\,\mathtt{x} = e\,]\!]_\delta & = & \mathtt{x} = [\![\,e\,]\!]_\delta \\
 & & \text{if}\ \ \mathtt{x} \notin \delta \\
[\![\,\mathtt{x} = e\,]\!]_\delta & = & \textbf{calib}(\mathtt{x} = [\![\,e\,]\!]_\delta) \\
 & & \text{if}\ \ \mathtt{x} \in \delta \\
[\![\,e.\mathtt{fd} = e'\,]\!]_\delta & = & \textbf{calib}([\![\,e\,]\!]_\delta.\mathtt{fd} = [\![\,e'\,]\!]_\delta)
\end{array}
$$

Fig. 24. Automated Calibration Insertion

and (R-VWrite) define local variable read/write. (R-New) defines object instantiation, and (R-Msg) addresses messaging (method invocation). (R-InE) captures execution within a closure, and (R-InV) defines closure destruction. (R-MPop) models S-stack pop, and (R-Cont) models continuation.

C. Automated Calibration Insertion

The operational semantics we defined earlier requires programmers explicitly place **calib** expressions in their programs to check sustainability. We now define a compiler transformation to automatically identify and insert calibration points.

To support automated calibration, several design choices are possible: (1) check periodically; (2) check whenever the supply gauge changes; (3) check whenever the demand gauge changes. For route (1), we find it challenging and *ad hoc* to set a fixed "period" that fits all programs. For (2), the supply gauge in energy-aware programming is often associated with system-level variables (such as remaining battery) whose updates are often performed outside the program runtime and out of the control of programmers. This may lead to platform-dependent behaviors: the same program may calibrate at different rates on two computers with different battery drivers. Eco chooses route (3). Demand gauge is typically an expression formed with program variables, such as loop index in the example in Figure 2. The state change of these variables is visible and predictable to the programmer. For application-level energy management strategies—a family that Eco belongs to— we believe the programmer should have control on program behaviors related to energy awareness.

The transformation is conceptually simple, defined in Figure 24. First, let us use metavariable δ to represent a set of variables, and helper function $\mathrm{FV}(e)$ computes the set of free variable names in e, a standard function we omit in this presentation. Function $[\![\,e\,]\!]_\delta$ transforms expression e given free variable set δ. Note that the body of a **uniform** block is not transformed, aligned with the fact that code in a uniform block should not trigger mode changes, and hence there is no need for calibration.

The last three cases are most relevant to our discussion here. If a variable is written/assigned, the calibration expression is only inserted if the variable also appears in the demand gauge of enclosing sustainable blocks. When the heap is mutated (through an object write), the calibration expression is always inserted, because heap mutation is known to have non-local effects. Evidently, the definition here is prohibitively conservative. Our treatment follows a standard path in language design and implementation: the formal system favors the most succinct way to illustrate high-level ideas, whereas redundant calibration removal is implemented as a compiler optimization (in Section VI).

$$(\text{T-Num})\ \Gamma \vdash n \xrightarrow{\textbf{num}} n$$

$$(\text{T-New})\ \Gamma \vdash \textbf{new}\ \mathtt{X} \xrightarrow{\mathtt{X}} \textbf{new}\ \mathtt{X}$$

$$(\text{T-VRead})\ \Gamma \vdash \mathtt{x} \xrightarrow{\Gamma(\mathtt{x})} \mathtt{x}$$

$$(\text{T-VWrite})\ \frac{\Gamma \vdash e \xrightarrow{\tau} e' \qquad (\tau \nearrow \Gamma(\mathtt{x}))\#(e' \nearrow e'')}{\Gamma \vdash \mathtt{x} = e \xrightarrow{\Gamma(\mathtt{x})} \mathtt{x} = e''}$$

$$(\text{T-FRead})\ \frac{\begin{array}{c}\Gamma \vdash e \xrightarrow{\tau} e' \qquad \mathtt{X} = \text{ATOM}(\tau) \\ (\tau \nearrow \mathtt{X})\#(e' \nearrow e'') \qquad (\tau_0\ \mathtt{fd}) \in \text{FIELDS}(\mathtt{X})\end{array}}{\Gamma \vdash e.\mathtt{fd} \xrightarrow{\tau_0} e''.\mathtt{fd}}$$

$$(\text{T-FWrite})\ \frac{\begin{array}{c}\Gamma \vdash e_1 \xrightarrow{\tau_1} e_1'' \\ \Gamma \vdash e_2 \xrightarrow{\tau_2} e_2'' \qquad \mathtt{X} = \text{ATOM}(\tau_1) \qquad (\tau_0\ \mathtt{fd}) \in \text{FIELDS}(\mathtt{X}) \\ (\tau_1 \nearrow \mathtt{X})\#(e_1'' \nearrow e_1') \qquad (\tau_2 \nearrow \tau_0)\#(e_2'' \nearrow e_2')\end{array}}{\Gamma \vdash (e_1.\mathtt{fd} = e_2) \xrightarrow{\tau_0} (e_1'.\mathtt{fd} = e_2')}$$

$$(\text{T-Msg})\ \frac{\begin{array}{c}\Gamma \vdash e_1 \xrightarrow{\tau_1} e_1'' \qquad \Gamma \vdash e_2 \xrightarrow{\tau_2} e_2'' \\ \mathtt{X} = \text{ATOM}(\tau_1) \qquad \text{MTYPE}(\mathtt{X}, \mathtt{md}) = \tau_2' \rightarrow \tau_0 \\ (\tau_1 \nearrow \mathtt{X})\#(e_1'' \nearrow e_1') \qquad (\tau_2 \nearrow \tau_2')\#(e_2'' \nearrow e_2')\end{array}}{\Gamma \vdash (e_1.\mathtt{md}(e_2)) \xrightarrow{\tau_0} (e_1'.\mathtt{md}(e_2'))}$$

$$(\text{T-Mcase})\ \frac{\Gamma \vdash e_i \xrightarrow{\tau} e_i'}{\Gamma \vdash \{\overline{\mathtt{m} : e}\}^\tau \xrightarrow{\textbf{mcase}\langle\tau\rangle} \{\overline{\mathtt{m} : e'}\}^\tau}$$

$$(\text{T-Cont})\ \frac{\Gamma \vdash e_1 \xrightarrow{\tau_1} e_1' \qquad \Gamma \vdash e_2 \xrightarrow{\tau_2} e_2'}{\Gamma \vdash e_1; e_2 \xrightarrow{\tau_2} e_1'; e_2'}$$

$$(\text{T-Sustain})\ \frac{\begin{array}{c}\Gamma \vdash e \xrightarrow{\tau} e' \qquad \Gamma \vdash e_i \xrightarrow{\tau_i} e_i'' \qquad \text{IM}(e_i) \\ (\tau_i \nearrow \textbf{num})\#(e_i'' \nearrow e_i') \qquad \text{for any } i = 1,2,3,4\end{array}}{\Gamma \vdash \begin{array}{l}\textbf{sustainable}\ \{e\}(\langle e_1; e_2; e_3; e_4\rangle) \xrightarrow{\tau} \\ \textbf{sustainable}\ \{e'\}(\langle e_1'; e_2'; e_3'; e_4'\rangle)\end{array}}$$

$$(\text{T-Uniform})\ \frac{\Gamma \vdash e \xrightarrow{\tau} e'}{\Gamma \vdash \textbf{uniform}\ \{e\} \xrightarrow{\tau} \textbf{uniform}\ \{e'\}}$$

$$
\begin{array}{rcl}
\text{ATOM}(\mathtt{X}) & \overset{\text{def}}{=} & \mathtt{X} \\
\text{ATOM}(\textbf{num}) & \overset{\text{def}}{=} & \textbf{num} \\
\text{ATOM}(\textbf{mcase}\langle\tau\rangle) & \overset{\text{def}}{=} & \text{ATOM}(\tau)
\end{array}
$$

Fig. 25. Type Checking and Type-Directed Mode Selection Insertion

D. Type Checking and Type-Directed Mode Selection Insertion

Eco is a strongly typed language. Subtyping relation <: is reflexive and transitive, and in addition $\tau <: \tau'$ holds if (1) $\tau = \mathtt{X}$, and $\tau' = \mathtt{Y}$, and **class** $\mathtt{X}$ **extends** $\mathtt{Y}$ $\{\overline{F}\ \overline{M}\}$ appears in the program. (2) $\tau = \textbf{mcase}\langle\tau_0\rangle$ and $\tau' = \textbf{mcase}\langle\tau_0'\rangle$ and $\tau_0 <: \tau_0'$. The first rule is standard Java nominal subtyping. The second rule says covariant subtyping for mode cases is supported. This is sound because individual case members in a mode case cannot be mutated once the mode case is constructed.

The type checking algorithm by itself contains few surprises. What makes the type system of Eco interesting is it can help programmers insert mode case **select** expressions automatically. This is beneficial for *incremental* energy-aware programming. A programmer wishing to enrich the Java program with energy-aware support may start by altering a variable from, say, holding a `Res(1024, 768)` object to the mode case we used in Figure 2. Without additional support, the programmer would have to insert **select** expressions explicitly for every occurrence of such variables in a sustainable block.

We define a unified process for both type checking and automated mode case selection insertion, in Figure 25. We use metavariable Γ to represent a *typing environment*, defined as a sequence of elements in the form of $\mathtt{x} : \tau$, saying variable $\mathtt{x}$ has type τ. We further use notation $\Gamma(\mathtt{x})$ to compute τ', where $\mathtt{x} : \tau'$ is the rightmost element in Γ for any τ'. Judgement $\Gamma \vdash e \xrightarrow{\tau} e'$ says expression e has type τ under typing environment Γ, and the expression can be transformed to e' where appropriate **select** expressions are inserted. The definition of the transformation is hinged upon the following definition:

$$\frac{\tau <: \tau'}{(\tau \nearrow \tau')\#(e \nearrow e)}$$

$$\frac{\neg(\tau <: \tau') \qquad \tau = \textbf{mcase}\langle\tau''\rangle \qquad (\tau'' \nearrow \tau')\#(e \nearrow e')}{(\tau \nearrow \tau')\#(e \nearrow \textbf{select}\ e')}$$

Predicate $(\tau \nearrow \tau')\#(e \nearrow e')$ says an expression e of type τ should be transformed to e' if it appears in a context expecting an expression of type τ'. Some examples should be sufficient to illustrate the ideas here:

$$(\textbf{mcase}\langle\textbf{num}\rangle \nearrow \textbf{num})\#(e \nearrow \textbf{select}\ e)$$
$$(\textbf{mcase}\langle\textbf{mcase}\langle\textbf{num}\rangle\rangle \nearrow \textbf{num})\#(e \nearrow \textbf{select}\ (\textbf{select}\ e))$$
$$(\textbf{mcase}\langle\mathtt{X}\rangle \nearrow \mathtt{Y})\#(e \nearrow \textbf{select}\ e) \quad \text{where} \quad \mathtt{X} <: \mathtt{Y}$$

Implicitly, the definition here (and the automatic insertion algorithm) supports the case of *higher-order* mode cases, where the case member of a mode case is another mode case.

The more interesting rules are (T-FRead), (T-FWrite), (T-Msg), (T-VWrite), and (T-Sustain). In the first 3 cases, the expected type for the (field read/field write/messaging) target must be an object type. If the target is a mode case, they need to be destructed. Take field read expression `x.fd` for example. If variable `x` turns out having type $\textbf{mcase}\langle\textbf{mcase}\langle\mathtt{X}\rangle\rangle$, the transformed expression should be (**select** (**select** `x`))`.fd`. For field write, messaging, and variable assignment, an additional

$$\text{(T-Program)}\ \frac{\overline{C \to C'} \qquad \text{TOTAL}(\overline{D}) \qquad [\,] \vdash e \xrightarrow{\tau} e' \text{ for some } \tau, e'}{\langle\overline{C}; \overline{D}; e\rangle \to \langle\overline{C'}; \overline{D}; [\![\, e' \,]\!]_{c_{\mathtt{boot}}}\rangle}$$

$$\text{(T-Cls)}\ \frac{\mathtt{X}, \mathtt{Y} \vdash \overline{M \to M'}}{\textbf{class}\ \mathtt{X}\ \textbf{extends}\ \mathtt{Y}\{\overline{F}\ \overline{M}\} \to \textbf{class}\ \mathtt{X}\ \textbf{extends}\ \mathtt{Y}\{\overline{F}\ \overline{M'}\}}$$

$$\text{(T-Md)}\ \frac{\begin{array}{c}\mathtt{x} : \tau', \textbf{this} : \mathtt{X}, \Gamma \vdash e \xrightarrow{\tau} e' \\ \text{MTYPE}(\mathtt{Y}, \mathtt{md}) = \tau_0' \to \tau_0 \text{ implies } \tau_0' = \tau' \land \tau_0 = \tau\end{array}}{\mathtt{X}, \mathtt{Y} \vdash \tau\ \mathtt{md}(\tau'\ \mathtt{x})\{\Gamma\ e\} \to \tau\ \mathtt{md}(\tau'\ \mathtt{x})\{\Gamma\ [\![\, e' \,]\!]_{c_{\mathtt{boot}}}\}}$$

Fig. 26. Class Typing

consideration is to match the types on both ends of the data flow. For instance, expression $\mathtt{x} = \mathtt{y}$ needs to be transformed to $\mathtt{x} = \textbf{select}\ \mathtt{y}$ if $\mathtt{x}$ has type $\textbf{mcase}\langle\mathtt{X}\rangle$ whereas $\mathtt{y}$ has type $\textbf{mcase}\langle\textbf{mcase}\langle\mathtt{X}\rangle\rangle$. As a counterexample, the judgment cannot be established (and hence a type error) if $\mathtt{y}$ has type $\textbf{mcase}\langle\mathtt{X}\rangle$ and $\mathtt{x}$ has type $\textbf{mcase}\langle\textbf{mcase}\langle\mathtt{X}\rangle\rangle$. The same scenario applies for data flows established when the field of an object is set in (T-FWrite), or the formal parameter of a method is assigned in (T-Msg).

Finally, the four expressions in the characterizer must be of numeric type. Otherwise, **select** expressions need to be inserted. In addition, we require the expressions that appear in the characterizer be effect-free. This is captured by predicate $\text{IM}(e)$, which holds iff e does not contain subexpressions that are either local assignment, field write, or messaging. (In Eco implementation, we relax this definition to allow for messaging expressions that do not produce side effects.) Recall in the previous section, we defined an algorithm to enable calibration — *i.e.*, the evaluation of the expressions in the characterizer — when effectful expressions are evaluated in a sustainable block. The $\text{IM}(e)$ condition we introduce here avoids the counter-intuitive case where the evaluation of the characterizer expressions produces effects by itself.

E. Whole-Program Typing, Transformation, and Bootstrapping

Finally, we define the typechecking and transformation on the entire program, defined in Figure 26. The definition is similar to FJ, except that both the method bodies and the bootstrapping expression are transformed to include **select** and **calib**. For convenience, we define $c_{\mathtt{boot}} = \langle 0; 0; 0; 0\rangle$. Finally, we can define program bootstrapping in full. Given program $P = \langle\overline{C}; \overline{D}; e\rangle$, the bootstrapping configuration is

$$\langle[\,]; \overline{\mathtt{x} \mapsto \textbf{null}}; c_{\mathtt{boot}} : \mathtt{m}_{\mathtt{boot}}; e'\rangle$$

where $\overline{\mathtt{x}} = \text{FV}(e)$, $\langle\overline{C}; \overline{D}; e\rangle \to \langle\overline{C'}; \overline{D}; e'\rangle$, $\mathtt{m}_{\mathtt{boot}}$ is the greatest value in $\overline{D}$, and the subsequent reductions are implicitly parameterized by $\overline{C'}$ and $\overline{D}$.