Energy Types

Michael Cohen Haitao Steve Zhu Senem Ezgi Emgin Yu David Liu

Department of Computer Science
SUNY Binghamton
Binghamton NY 13902, USA
{mcohen3,hzhu1,semgin1,davidL}@binghamton.edu

Abstract

This paper presents a novel type system to promote and facilitate energy-aware programming. *Energy Types* is built upon a key insight into today's energy-efficient systems and applications: despite the popular perception that energy and power can only be described in joules and watts, real-world energy management is often based on discrete *phases* and *modes*, which in turn can be reasoned about by type systems very effectively. A phase characterizes a distinct pattern of program workload, and a mode represents an energy state the program is expected to execute in.

This paper describes a programming model where phases and modes can be intuitively specified by programmers or inferred by the compiler as type information. It demonstrates how a type-based approach to reasoning about phases and modes can help promote energy efficiency. The soundness of our type system and the invariants related to inter-phase and inter-mode interactions are rigorously proved. Energy Types is implemented as the core of a prototyped object-oriented language ET for smartphone programming. Preliminary studies show ET can lead to significant energy savings for Android Apps.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; C.0 [*Computer Systems Organization*]: General—Hardware/Software Interfaces

Keywords Energy Efficiency, Energy-Aware Software, Type Systems

1. Introduction

There is a long history of designing type systems to improve software quality, such as robustness, reliability, and security. As we look forward, energy efficiency is increasingly looming large as a critical software metric. From cloud computing servers, to sensor networks, to iPads and Droids, new hardware platforms redefine what "acceptable software" is – occasional crashes may be tolerated, but unacceptable are electricity bills in an astronomical amount for a data center, or battery life of just a few hours for a smartphone. Prior research on energy efficiency concentrates on innovations in VLSI (*e.g.* [7]), architectures (*e.g.* [17]), operating systems (*e.g.* [38]), and compiler optimizations (*e.g.* [18]). A much less explored path is how innovations on programming models [4, 30, 34] can help build energy-friendly software, and in particular, how programming language technologies can help *reason about energy management*.

To some extent, the lack of development in reasoning energy-aware software is understandable. First, energy is measured in joules and watts – continuous values (or their floating-point number representations) that are often unfriendly to reasoning techniques. Second, energy consumption is a combined effect of many hardware components – such as CPUs, caches, DRAMs, I/O devices – and they often interact in complex ways so that energy consumption is impossible to precisely estimate. Third, unlike lower-layer solutions where effectiveness can be quantified unambiguously – *e.g.* "this CMOS circuit contributes to an energy saving of 0.31 μW." – innovations on language designs assume a rational programmer, and the effectiveness depends on the programmer and the nature of the program.

This paper presents Energy Types, a practical type system to reason about energy-aware software. Instead of directly reasoning about joules and watts over complex hardware – an ambitious task perhaps impossible to get perfectly right and practical – Energy Types reasons about *phases* and *modes*, two recurring motifs in modern energy-aware software.

1.1 Phases as Types

An established fact in architecture research is that a program usually demonstrates *phased behaviors* of energy consumption ([8, 17]): the rate of energy consumption (*i.e.* power) flows and ebbs in phases, steady within but drastically differ-

OOPSLA'12, October 19–26, 2012, Tucson, Arizona, USA.

ent across. Intuitively, program fragments with different logical goals have distinct workloads, which yield distinct patterns of CPU usage, memory accesses, cache misses, and I/O operations, and eventually lead to a distinct pattern of energy consumption. Program phase characterization – determining the number of phases, and the boundary of each – provides important insight into understanding program energy consumption behaviors, and further guides energy management techniques for better decision-making. Existing phase prediction techniques (*e.g.* [16, 31]) usually combine dynamic monitoring and profiling with sophisticated classification or prediction algorithms.

Phase Support in Energy Types Energy Types brings the knowledge of programmers into the equation. We argue that a programmer often intuitively knows the logical sub-goals of her program, which often corresponds to phases with distinct patterns of energy consumption. For example, it is not difficult for a programmer to realize that her Mars landing game is composed of a (CPU-intensive) physics calculation phase and an (I/O-intensive) 3D rendering phase.

Programmers in Energy Types can label data and operations with *phase type qualifiers*, directly specifying phased behaviors by relating computational elements to phases of energy consumption. Energy Types helps programmers preserve a consistent view of phase placement by enforcing *phase distinction*: each data or operation must commit to – *i.e.* be typed with – one and only one phase. The type system further enforces *phase isolation*: any cross-phase interaction must be accompanied with explicit type coercion.

There are two important energy efficiency benefits that result from building phases into an energy-aware programming language. First, hardware-level energy management procedures often need to adjust the status of hardware (to save energy) based on the status of the software runtime. Phase type qualifiers are invaluable at this software/hardware interface:

ENERGY MANAGEMENT BENEFIT 1 (Guiding Hardware through Types). Phase type information can be used to guide hardware-level energy management, because it taps both the knowledge of the programmer through type declaration, and the ability of a type system to propagate phase information.

Concretely, the host language of Energy Types – ET – is equipped with a dynamic semantics that supports CPU dynamic voltage and frequency scaling (DVFS) [29] directed by phase type information. As one of the most time-honored hardware-level energy management strategies, DVFS is a trade-off between energy and speed. It is most advantageous to scale down the CPU frequency, so that energy can be saved, but only at a time when the CPU is least busy, so that performance degrades the least. Its effectiveness is largely dependent upon choosing the right scaling point ("where to scale") and the right scaling factor ("what frequency/voltage to scale to"). We offer a language design, accompanied by an experimental validation, to demonstrate phases as types can provide answers to both questions that yield energy savings.

Thanks to the type system enforcement of phase distinction and phase isolation, phases as types further *promote* phased behaviors. In Energy Types, intra-phase object interaction – say messaging – is supported in the same syntax as in the familiar Java model, whereas any inter-phase interaction requires explicit type coercion, an inconvenience programmers would rather avoid. Given the requirement of phase isolation, the programmer is motivated to re-factor the code to cluster calculation operations separately from rendering operations, instead of intertwining code with distinct patterns of system usages. To summarize:

ENERGY MANAGEMENT BENEFIT 2 (Promoting Phased Behaviors). Phase distinction and phase isolation promote the construction of programs with more stable power consumption within phases and fewer phase changes, both beneficial for phase-based energy management.

1.2 Modes as Types

Application-specific energy savings are another important source for software energy efficiency. It has been observed that [4, 26, 30, 34] applications can often be programmed in different ways. All are acceptable, but different choices may consume different levels of energy and be best used in different energy states. For example, a programmer may decide to render a high-fidelity Mars lander image when her smartphone is fully charged, and only a low-fidelity image when the battery level of the phone is below 20%.

Mode Support in Energy Types Energy Types builds on this insight, and offers programmers the capability to associate data and operations with *mode type qualifiers*. Each mode is a programmer-defined and typed energy state, indicating the expected energy usage context associated with specific data or operations, such as "battery high" and "battery low." Allowing for programmer mode declarations is in sync with our philosophy on constructing a *practical* static reasoning system for energy management: instead of painstakingly determining why rendering a high-fidelity image consumes more energy than a low-fidelity image through advanced static analysis, we believe programmers intuitively know. The goal of Energy Types is thus to assist rational yet imperfect programmers to consistently assign energy consumption expectations to program elements.

Building modes into an energy-aware programming language leads to two benefits toward energy efficiency:

ENERGY MANAGEMENT BENEFIT 3 (Encouraging Application-Specific Energy Savings). The process of thinking how many modes should be supported, and how data and operations should be assigned to each mode is the same as designing application-specific energy-saving strategies.

ENERGY MANAGEMENT BENEFIT 4 (Regulating Energy States). The type-based approach regulates what program fragments can be used under different energy states, which on the high level is aligned with the programmer intuition of constructing energy-aware programs friendly to distinct energy states.

To achieve Benefit 4, our type system enforces a *waterfall invariant*: a program element associated with a "higher" energy state – one with more availability for energy consumption – may access an element with a "lower" energy state, but *not vice versa*. This design decision is driven by the requirement of energy-aware programming: invoking a function associated with a "higher" energy state – often more energy-consuming – when the system is in a low energy state may drain the system energy source, an *energy error*.

1.3 Energy Types

Energy Types is a foundational study of the fabrics of energy-aware software, abstracting the recurring themes of energy management as the reasoning of phases and modes. The type system and its host language ET are one of the first efforts to build common energy management strategies *directly into a programming language*, and *guarantee the consistency* of the programmers' energy management intentions during their practice of energy-aware programming.

In addition to Benefits 1-4, Energy Types inherits some benefits from type system solutions which may be appealing for energy-efficient computing in general: it is portable and platform independent; it promotes compositional reasoning of energy management behaviors; and it serves as lightweight and consistent documentation throughout the often iterative process of energy optimization.

This paper makes the following contributions:

- It introduces one of the first systematic studies on static reasoning for energy-aware software, in the shape of a type system that has been proven sound;
- It formulates invariants related to recurring themes of energy management as properties of phases and modes, with their enforcement on the language level formally proved;
- It provides a small-step semantics that demonstrates how traditional energy-saving strategies such as DVFS can be type-directed, and a formal illustration on type-based scaling factor selection;
- It offers an easy-to-use programming model that has been implemented as a prototyped compiler, and evaluated through programming Android Apps.

The rest of the paper is organized as follows. Sec.2 is an informal discussion on Energy Types and ET. The formal Energy Types system is presented in Sec. 3 and the implementation of ET and its evaluation is presented in Sec. 4. The last sections describe related work and future work.

```
1  phases { graphics <cpu main; main <cpu math; }
2  modes {hifi <: full; lofi <: hifi; }
3  class Main {
4    main () {
5      Recognizer rz = new Recognizer();
6      View  v = adapt (new View());
7      while(true) {
8        Gesture g = processInput();
9        int result = rz.recognize(g);
10       v.paintOverlay(adapt[graphics] g, result);
11     }
12   }
13   Gesture processInput()
14   { ...
15     return new Gesture@phase(math)();
16   }
17 }
18 class Recognizer {
19   Matcher@mode(full) m1 = newM(0.995);
20   Matcher@mode(hifi) m2 = newM(0.9);
21   Matcher@mode(lofi) m3 = newM(0.5);
22   DistCal d1 = new Cosine@mode(hifi)();
23   DistCal d2 = new Euclidean@mode(lofi)();
24   int recognize(Gesture g) {
25     mswitch(batteryState()) {
26       case full: return m1.match(g, d1);
27       case hifi: return m2.match(g, d2);
28       case lofi: return m3.match(g, d2);
29     }
30   }
31   Matcher newM(double p) {
32      return adapt (new Matcher(p));
33   }
34   modev batteryState() {
35       if(Util.batterycharged()) return full;
36       if (Util.batterylevel() > 0.3)
37          return hifi
38       else
39          return lofi;
40   }
41 }
42 class Matcher@phase(math) {
43    double precision;
44    Gesture[] ps = ...; //recognizable patterns
45    Matcher(double precision)
46      { this.precision = precision; }
47    int match (Gesture g, DistCal dc) {
48       Gesture gs = sampling(g, precision);
49       for (int i = 0; i< ps.length; i++)
50         {if( dc.compute(gs, ps[i]) < THRES )
51             return i;
52         }
53    }
54    ...
55 }
56 class View@phase(graphics) {
57    Gesture[] p = ...; //recognizable patterns
58    void paintOverlay(Gesture g, int result) {
59        paint(g);
60        paint(p[result]);
61    }
62    void paint(Gesture g) {
63       Util.draw(g.Coordinates());
64    }
65 }
66 class DistCal {...}
67 class Cosine extends DistCal {...}
68 class Euclidean extends DistCal {...}
69 class Gesture {...}
70 static class Util {
71    boolean batterycharged() { ... }
72    double batterylevel() { ... }
73    void draw(int co) { }
74 }
```

Figure 1. A Gesture Recognition Program in ET

2. Energy-Aware Programming in ET

ET is a Java-like object-oriented language for smartphone programming – an immediate testing ground we choose for Energy Types. A sample ET program is illustrated in Fig. 1. Key language features are highlighted in red.

This program, inspired by `android.gesture` APIs, applies pattern recognition techniques to determine what "gesture" the fingers of a smartphone user perform on the touch screen. The core program logic follows the predictable "input-recognize-render" sequence, in L. 8 - 10. The recognition logic is encapsulated in the `Recognizer` class, which selects different recognition algorithms based on the battery state of the smartphone (L. 25 - 29). The mathematical core of the recognition is implemented by class `Matcher`, which can be instantiated with different recognition `precision` settings and different mathematical routines to compute distances in a vector space, such as either in `Cosine` distance or in `Euclidean` distance.

The Pre-ET Era To motivate the need for ET, let us imagine what a green-conscious Android Java programmer would do in her attempt of energy-aware programming.

First, an informed programmer – especially one from architecture/VLSI/OS communities – *might* know that programs of different workloads often react to CPU scaling differently, in that scaling down CPU frequency/voltage of a "I/O-bound" part of the program such as the `paintOverlay` method, may lead to energy savings with little performance penalty. Her plan thus is to insert a pair of DVFS system calls in the source code, one before the messaging on L. 10 to scale down the CPU (say she decides to use 300Mhz) and then the other right after the messaging to scale the CPU back up to the previous level. The ensuing program might be faced with a number of usability, efficiency, and robustness issues, in that such DVFS calls might be:

- platform-dependent, *e.g.* the same program might be deployed to a machine that does not support 300Mhz
- redundant, *e.g.* both the message sender and the receiver insert the same DVFS calls to "err on the safe side"
- unbalanced, *e.g.* DVFS is used to scale the CPU down before `paintOverlay` messaging, but forgotten to be used afterwards to scale the CPU back up
- inconsistent *e.g.* the `View` object is thought to be I/O intensive at some program point, but may flow to a variable used as if it were CPU-intensive

Second, a green-conscious programmer may also apply to application-specific savings, such as creating different `Matcher` objects – similar to L. 19 - L.21 but without the qualifiers – each for a different battery level. What is required for her is to memorize the energy consumption traits for individual program fragments – `m1`, `m2`, `m3` in the program – and hopefully not accidentally use a `m1` or `m2` object when the system is low on battery. This task may appear to be simple for a program as short as Fig. 1, but not obviously so when the program grows larger, where objects references are routinely passed around, stored on the heap, and aliased.

This is not to mention the fact that the difficulties described above are indeed the *privileges* of advanced Android programmers. The majority of Apps programmers are not even aware of the options of energy-aware programming in the first place. It is these kinds of problems that ET addresses.

Phases and Modes The most noticeable features of ET are the optional type qualifiers of phases and modes.

Instead of thinking how low-level CPU frequencies should be scaled, ET programmers are encouraged to think how CPU-intensive each program fragment is. Either a class (L. 56) or an object (L. 15) can be associated with a phase qualifier to characterize such intensity. In addition, ET programmers can declare a partial order `<cpu`, meaning "less CPU-intensive than," through the **phases** declaration. For example, L. 1 says that phase `graphics` is less CPU-intensive than the phase of the bootstrapping code with a built-in phase named `main`, which in turn is less CPU-intensive than phase `math`. This relation encourages programmers to contribute in their knowledge, so that DVFS calls are inserted automatically, and the decision of scaling down/up is conducted by the compiler based on the partial order. Based on our experience, aligning a logical program unit with distinct CPU characteristics is not hard. For instance, the graphics/math differentiation here is in fact a common pattern in game programming. In Sec. 3.3, we show how this partial order may affect our choice of frequency scaling factors.

In addition, the ET programmer does not need to implicitly memorize that `m2` is more energy-consuming than `m3`. She instead can explicitly indicate in L. 20 that the `m2` object has a mode called `hifi` and in L. 21 that the `m3` object there has a mode called `lofi`. In addition, by using the **modes** construct in L. 2, the programmer indicates that objects qualified by `lofi` should be used when a lower energy consumption is expected. Indeed, modes naturally fall into a lattice, each indicating a level of expected energy consumption. The `<:` relation specified in **modes** construct is a subtyping relation. This can be explained in the same fashion where the waterfall invariant was explained in Sec. 1: when the program is expecting to interact with a high energy-consumption object, it is OK to end up interacting with a low energy-consumption object, but not *vice versa*.

The run-time manifestation of a phase is a portion of the execution with a relatively stable yet distinct intensity of CPU usage. The run-time manifestation of a mode is a portion of the execution specifically for a particular energy state. When they are represented in programs, especially in an object-oriented setting, both are represented by a network

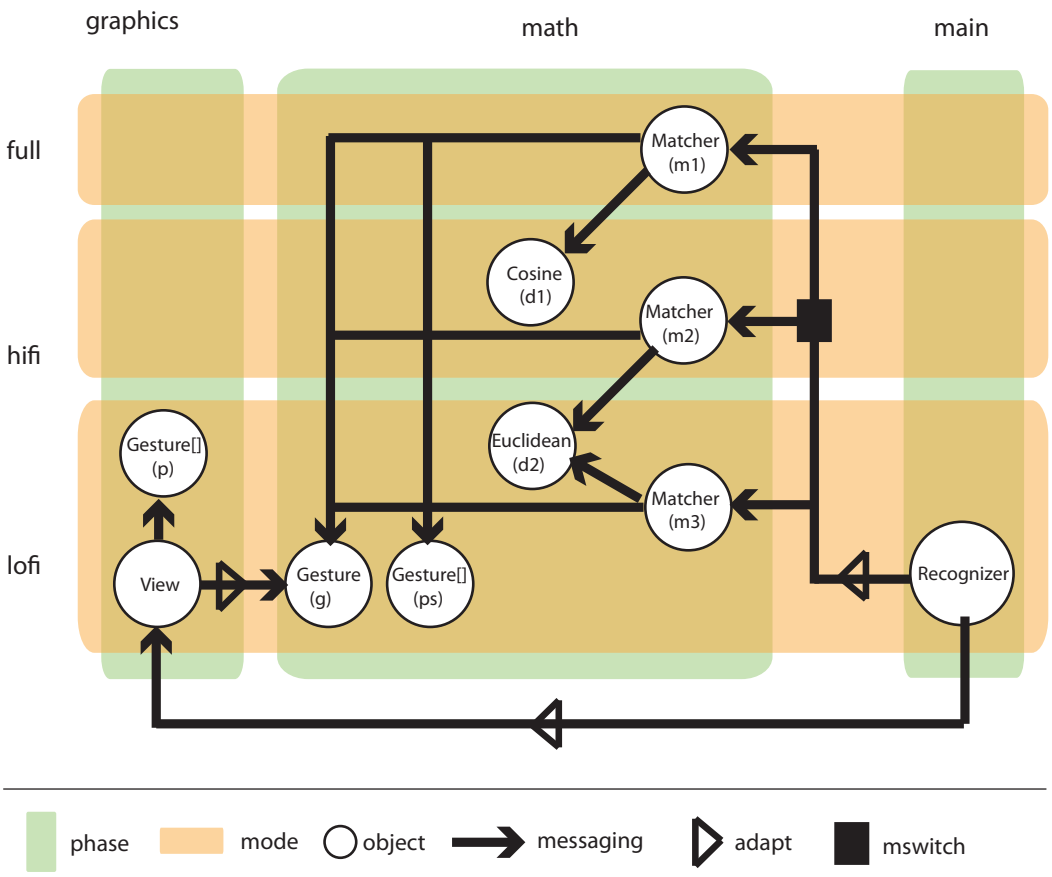

Figure 2. An Illustration of the Gesture Recognition program in Fig. 1

of objects interacting with each other. This view does not lie far apart from the foundational view of object orientation: each object is an encapsulation of data and their related operations; together, they are composed as a program, in the form of a network of messages. In that light, by labeling an *object* with a particular phase or mode qualifier, we are not only labeling *data*, but also (perhaps more importantly) defining the nature of the *interactions* the object is participating in through sending/receiving messages. It is these object interactions we need to investigate in more detail next.

"Healthy" Interactions Overall, our type system reasons about phases and modes through a core reminiscent of region types [36], with additional invariants on "healthy" inter-phase and inter-mode interaction. To set the analogy with region types further, Energy Types on the high level can be imagined as assigning objects (or expressions that represent them) to the "intersection" of a phase region and a mode region. Such declarations "pigeonhole" objects. Each object – by declaration or inference – lives in the intersection of one mode "hole" and one phase "hole", as seen in Fig. 2. What is more unique to Energy Types is how objects in these pigeonholes interact "healthily" in the presence of energy-aware programming.

The region type formulation provides natural support for both phase distinction and phase isolation we mentioned in Sec. 1. As a result, an object may freely access other objects belonging to the same phase (such as messaging and field access) – their interactions together form the execution sequence with stable CPU usage – but when cross-phase access is needed, programmers need to explicitly allow it. This in ET is represented as a phase coercion operator, the **adapt** expression. For example in L. 10, a `Gesture` object `g` is adapted to the `graphics` phase, so that it can be accessed freely within the `paintOverlay` method of an object of `graphics` phase. A common pattern of adaptation is to adapt an object to the phase of the enclosing object. As a convenience, we allow programmers to elide the phase name in this case. L. 32 is one such example. Adaptation in ET is static type coercion with no run-time overhead.

Similarly, objects with the same mode can also interact freely – their interactions together form the execution sequence specifically prepared for an energy state of the system. For objects with different modes to interact, the water-

fall invariant we introduced in Sec. 1 applies. For example in Fig. 2, the `Matcher` object `m2` is sending a message to the `Euclidean` object `d2`, but the former is of mode `hifi`, whereas the latter is of mode `lofi`.

Without additional support, a program subject to the waterfall invariant would "drop" further and further to a "lower" mode. This clearly does not align with our notion that smartphones are rechargeable, and a large number of today's devices are powered by renewable power sources. The **mswitch** expression is designed to help programmers "re-align" the system energy state with the mode of current context – a type coercion on the mode of the context. Based on the value **mswitch** guards on, the mode of the enclosing context can be coerced to a new one. For instance, L. 25-29 coerces the mode of the context to either `full`, `hifi`, or `lofi` based on the return value of `batteryState`. To facilitate this programming pattern, we allow modes to be values, and their type is **modev**, as in L. 34.

Overall, on the spectrum of inter-phase/mode interactions, where "no interaction" and "arbitrary interaction" on two ends, Energy Types takes a middle-of-the-road model: no interaction unless explicitly specified. This is more pronounced for phases, where cross-phase messaging requires an **adapt** – a "hassle" that programmers try to avoid but cannot live without because phase isolation otherwise would dissect the program into disjoint components. For modes, cross-mode messaging from a "higher" energy state to a "lower" one is implicitly allowed thanks to the waterfall invariant, but not the other way around. **mswitch** plays the role of enabling messaging in that direction, but only when programmers explicitly intend so.

Inference We believe green-conscious programmers *are* receptive to changes to their coding habits, but may only be willing to do so if changes are minimalistic and incremental. This belief influenced our implemented language ET to take the *implicit* form of parametric polymorphism as default. Type information such as `@phase` or `@mode` declarations can appear anywhere a type may appear, but can be elided anywhere a programmer chooses to. In this light, ET can be viewed as a polymorphic system heavy on type inference but light on declaration. To see parametric polymorphism at work, note that our algorithm is able to analyze whether `dc.compute` in L. 50 conforms to the waterfall invariant separately for the three invocations at L. 26-28.

Inference is a double-edged sword with standard tradeoffs: the more helpful the language is able to save programmer annotation efforts, the less helpful it is to produce modular program specifications. It is our belief that in the context of energy-aware programming – a nascent practice where any additional programming task is *overhead* for average Apps programmers – we as language designers should start with solutions minimalistic in syntax. As an effort to separate this implementation decision from foundations, the formal type system presented in this paper is based on the more standard explicit form of parametric polymorphism to elucidate ideas. Energy Types with inference is fully formalized in a technical report [37].

Formal Guarantees Energy Types is a *sound* constraint-based type system. Translating this to energy management, it ensures that a consistent view of the programmer's energy characterization and energy management intentions is maintained. For instance, it would lead to a type error if a programmer in one part of the program characterizes an object as phase `graphics` mode `hifi`, but – in another part of the program – forgets and uses the same object as if it belonged to phase `math` mode `lofi`.

Type preservation (subject reduction) from a temporal angle tells an intuitive fact about what a program should behave: its execution should be *stable* in maintaining its phase and mode characterization (the types), except when explicit phase change or mode change happens.

A typechecked ET program is free of two kinds of errors: (1) an object accesses (field access or messaging) to another object belonging to a different phase without explicit **adapt**; (2) an object sends a message to another object in a "higher" mode without explicit **mswitch**; The details of these guarantees will be elaborated in Sec. 3.4.

Unfortunately but predictably, Energy Types does not have the *guarantee* of saving energy. Indeed, no magical technologies – be it out of computer science, chemical engineering, material science, or sub-atomic physics – can *guarantee* energy savings. The importance of ET is to provide formal guarantees to make sure the principles of energy management are promoted and abided by, which in turn produces high-quality energy-aware software with effective energy management, and ultimately leads to energy savings.

ET in the Context of Energy-Efficient Computing The innate drawback of a programming language approach – especially through the lens of solutions from lower compute stacks – is that it can never prevent a *rogue* programmer. For instance, a rogue programmer, for the sake of proving a point, could intentionally declare `lofi` to an object known to lead to heavy energy consumption, and systematically use this object as if it consumed little energy. This is analogous to scenarios such as an information flow language user intentionally declares a password string with a low-security label.

ET is however an asset to a good-intentioned but imperfect programmer. Benefit 1 allows programmers to express their high-level knowledge relevant to low-level energy management. Benefit 2 encourages programmers to refactor their code to be friendly for energy management. Benefit 3 allows programmers to write code adaptive to different energy state. Benefit 4 discourages the imperfect programmers from writing code that may unnecessarily drain energy. Overall, the type system provides formal guarantees for effective energy-aware programming.

The roles of ET and existing dynamic approaches are complementary. To construct a primitive feedback loop from

Ψ	::=	$\overline{PO}\ \overline{MO}\ \overline{C}$	*program*
PO	::=	$\mathtt{p} <_{\mathtt{cpu}} \mathtt{p}'$	*phase decl*
MO	::=	$\mathtt{m} <: \mathtt{m}'$	*mode decl*
C	::=	**class** $\mathtt{c}\ \Delta$ **extends** $\tau\{K\ \overline{F}\ \overline{M}\ \}$	*class*
K	::=	$\mathtt{c}(\overline{\tau}\ \overline{\mathtt{fd}})\{\textbf{super}(\overline{\mathtt{fd}}); \textbf{this}.\overline{\mathtt{fd}} := \overline{\mathtt{fd}}\}$	*constructor*
F	::=	$\tau\ \mathtt{fd}$	*field*
M	::=	$\Delta\ \tau\ \mathtt{md}(\overline{\tau}\ \overline{\mathtt{x}})\{e\}$	*method*
e	::=	$\mathtt{x} \mid e.\mathtt{fd} \mid \textbf{new}\ \tau(\overline{e})$	*expressions*
	|	$e.\mathtt{md}\langle\iota\rangle(\overline{e}) \mid (\tau)e \mid \textbf{adapt}[\phi]e$	
	|	$\textbf{mswitch}\ (e)\{\overline{\mathtt{m}} : \overline{e}, e'\}$	
p	$\in$	PCONST	*phase name*
m	$\in$	MCONST	*mode name*
c	$\in$	$\mathsf{CN} \cup \{\mathtt{Object}, \mathtt{Main}\}$	*class name*
md	$\in$	$\mathsf{MN} \cup \{\mathtt{main}\}$	*method name*
fd			*field name*
x	$\in$	VAR	*variable name*

Figure 3. Energy Types Abstract Syntax

dynamics, imagine an iterative development process where a programmer incrementally adds phase and mode declarations to a program, each after running an energy profiler to help her make decisions. From that perspective, the mild level of annotations of ET can help profiler users consistently document the traits of energy consumption, following the "types as documentation" slogan. On the flip side, a programming language approach can broaden the optimization space of energy efficiency by bringing in the programmer knowledge into the equation. Overall, the emphasis of this paper is to offer a *type system* approach to look at energy management, not to trumpet a purely *static* approach. It is interesting future work to see whether Energy Types can be extended with profile-guided typing [11] or hybrid typing [2, 3, 20, 32] to integrate the static and the dynamic.

3. The Formal System

We now present Energy Types, the formal core of ET.

3.1 Abstract Syntax and Preliminaries

The abstract syntax is defined in Fig. 3 where notation $\overline{X}$ represents a sequence $X_1, \ldots, X_n$ of some n. A program Ψ is defined as phase ordering declarations $\overline{PO}$, mode ordering declarations $\overline{MO}$, as well as classes $\overline{C}$. The first two parts correspond to the **phases** and **modes** declarations in the concrete syntax. The formalized object-oriented features resemble Featherweight Java [14]. We follow their convention of using the highly stylized constructor definition (K), encoding assignment and statement sequencing as local method invocations, omitting field update, and choosing `Object` as the name of the inheritance root. In the **mswitch** $(e)\{\overline{\mathtt{m}} : \overline{e}, e'\}$ expression, $\overline{\mathtt{m}} : \overline{e}$ are all the **case**'s defined in the concrete syntax; e' is the "default" case when none of $\overline{\mathtt{m}}$ matches the value of e. The bootstrapping expression is enclosed in class `Main` and method `main`. We use metavariables `c`, `md`, `fd`, `x`, `p`, and `m` to represent names of classes, methods, fields, variables, phases, and modes. **this** $\in$ VAR. Type-related elements, such as τ and Δ, will be formally defined shortly in Sec. 3.2.

We choose to formalize an explicit form of parametric polymorphism in this short presentation. As a result, the abstract syntax here carries additional type annotations – such as Δ (whose definition we shall see soon) – that the implemented language does not need. The gap between the syntax here and the programmer syntax is filled by a polymorphic type inference, which has been implemented in ET. We describe this relatively independent topic toward the end of Sec. 3.2. This presentation choice also allows our formal system to bear greater resemblance to other classic systems of parametric polymorphism in object-oriented languages – such as Featherweight Generic Java (FGJ) – so that our discussion can be more focused on elucidating energy-related features.

Notations For any binary relation R, we use $\mathtt{dom}(R)$ to represent its domain, $\mathtt{range}(R)$ to represent its range, $\mathtt{whole}(R)$ to represent the union of the domain and the range, $(R)^*$ as the reflexive and transitive closure of R, and $R\ |_S$ to represent the restriction of binary relation R to set S.

The ubiquitous X, Y notation is used for concatenation of sequences X and Y. The concatenation is "flat" in that we do not create sequences of sequences. We use ϵ to represent an empty sequence, and the element itself to represent a (degenerate) 1-element sequence. When no confusion arises, we liberally treat a sequence as a set, and apply common operators such as $\in$, $\subseteq$. We call a sequence in the form of $[a_1 \mapsto b_1, \ldots a_n \mapsto b_n]$ as a mapping sequence. When order does not matter, we also view a mapping sequence as a binary relation, *i.e.* $\{(a_1, b_1), \ldots, (a_n, b_n)\}$. Given two mapping sequences X and Y, binary operation $X \uplus Y$ is defined as X, Y if $\mathtt{dom}(X) \cap \mathtt{dom}(Y) = \emptyset$; it is undefined otherwise.

3.2 The Type System

Types Type-related elements are defined in Fig. 4. A type can either be an object type $\mathtt{c}\langle\iota\rangle$ or a primitive type **modev** for mode values. If we take the standard view of a type as a characterization of values, the particular form of $\mathtt{c}\langle\iota\rangle$ aligns with our high-level intuition that any two objects instantiated from the same class `c` may have drastically different energy consumption characteristics, depending on how they are initialized, and what objects they interact with. Construct ι captures this notion, with each element in its phase sublist ($\overline{\phi}$) describing the phase characteristics of an object the current object interacts with, and each element in its mode sublist ($\overline{\mu}$) describing the mode characteristics. As a special case, the first element in $\overline{\phi}$ and that of $\overline{\mu}$ describe the phase and mode of the current object. For that purpose,

τ	::=	$\mathtt{c}\langle\iota\rangle \mid \textbf{modev}$	*type*
ι	::=	$\overline{\phi};\overline{\mu}$	*energy characterization params*
Ξ	::=	$\langle\phi;\mu\rangle$	*context object characterization*
ϕ	::=	$\mathtt{p} \mid \mathtt{pt}$	*phase*
μ	::=	$\mathtt{m} \mid \mathtt{mt}$	*mode*
$\mathtt{pt}$			*phase type variable*
$\mathtt{mt}$			*mode type variable*
Σ	::=	$\Theta;\Omega$	*constraints*
Θ	::=	$\overline{\phi \cong \phi'}$	*phase constraints*
Ω	::=	$\overline{\mu <: \mu'}$	*mode constraints*
Δ	::=	$\iota;\Sigma$	*energy specification*

Figure 4. Type Elements

we use metavariable Ξ to represent the energy characterization of the current object, and further define $\mathtt{ethis}(\iota) \stackrel{\text{def}}{=} \langle\phi;\mu\rangle$ where $\iota = \langle\phi,\overline{\phi};\mu,\overline{\mu}\rangle$. For example, a declared type `Matcher @phase(maths) @mode(hifi)` in the source code can be encoded in the formal system as:

$$\mathtt{Matcher}\langle\mathtt{maths},\phi_1,\ldots,\phi_m;\mathtt{hifi},\mu_1,\ldots,\mu_n\rangle$$

where $\phi_1,\ldots,\phi_m$ and $\mu_1,\ldots,\mu_n$ are characterizations for objects the `Matcher` object interacts with (inferred by the ET compiler). Type-theoretically, this syntactical form indicates that Energy Types belongs to the family of System F type systems (*e.g.* [14, 27, 36]). The non-trivial observation in this particular instance is that *energy consumption for code is fundamentally parametric*.

Type Constraints Unlike System $F_{<:}$ systems (*e.g.* FGJ) where constraints come only in one form (associating type variables with upper bounds), the type constraints of Energy Types come in two forms to reflect the different natures of phases and modes. Constraint $\phi \cong \phi'$ says ϕ and ϕ' must be "unified," intuitively meaning they must be representations of the same programmer-declared phase in $\overline{PO}$. Constraint $\mu <: \mu'$ says μ must represent a mode less energy-consuming than that of μ'. Sets Θ and Ω are used to hold the two kinds of constraints respectively and we use Σ to represent both.

Energy Specification The combination of energy characterization parameters (ι) and the constraints over them (Σ) form the specification of an object's *energy specification*, represented by metavariable Δ. When Δ appears in the class definition C, its energy characterization parameters is composed of type variables – `pt` for phases and `mt` for modes – which serve as type parameters that can be "customized" for different instances of the same class, the instantiation in parametric polymorphism. Definition $\Delta_1 \vdash_{\mathtt{inst}} \iota \downarrow \Delta_2$ says that parametric energy specification Δ_2 can be instantiated by energy characterization parameters ι under energy specification Δ_1.

$$\text{(WF-TClass)}\ \frac{\textbf{class}\ \mathtt{c}\ \Delta' \cdots \in \Psi \qquad \Delta \vdash_{\mathtt{inst}} \iota \downarrow \Delta'}{\Delta \vdash_{\mathtt{wft}} \mathtt{c}\langle\iota\rangle}$$

$$\text{(WF-TTop)}\ \frac{\iota = \phi;\mu \qquad \Delta \vdash_{\mathtt{inst}} \iota \downarrow \epsilon;\emptyset;\emptyset}{\Delta \vdash_{\mathtt{wft}} \mathtt{Object}\langle\iota\rangle}$$

$$\text{(WF-TModev)}\ \Delta \vdash_{\mathtt{wft}} \textbf{modev}$$

Figure 5. Type Well-Formedness

$$\text{(WF-PSet)}\ \frac{\begin{array}{c}\Psi = \overline{PO}\ \overline{MO}\ \overline{C} \\ \Theta \text{ reflexive and transitive} \\ \Theta \mid_{\mathsf{PCONST}} \text{identity} \\ \mathtt{whole}(\Theta) \subseteq \mathtt{whole}(\overline{PO}) \cup \overline{\phi}\end{array}}{\overline{\phi};\overline{\mu} \vdash_{\mathtt{wfp}} \Theta}$$

$$\text{(WF-MSet)}\ \frac{\begin{array}{c}\Psi = \overline{PO}\ \overline{MO}\ \overline{C} \\ \Omega \text{ reflexive and transitive} \\ \overline{MO} = \Omega \mid_{\mathsf{MCONST}} \\ \mathtt{whole}(\Omega) \subseteq \mathtt{whole}(\overline{MO}) \cup \overline{\mu}\end{array}}{\overline{\phi};\overline{\mu} \vdash_{\mathtt{wfm}} \Omega}$$

$$\text{(WF-Spec)}\ \frac{\iota \vdash_{\mathtt{wfp}} \Theta \qquad \iota \vdash_{\mathtt{wfm}} \Omega}{\vdash_{\mathtt{wfs}} \iota;\Theta;\Omega}$$

Figure 6. Energy Specification Well-Formedness

$$\frac{\begin{array}{c}\Delta_i = \iota_i;\Theta_i;\Omega_i \\ \iota \Subset \iota_1 \qquad \Theta_2\{\iota/\iota_2\} \subseteq \Theta_1 \qquad \Omega_2\{\iota/\iota_2\} \subseteq \Omega_1\end{array}}{\Delta_1 \vdash_{\mathtt{inst}} \iota \downarrow \Delta_2}$$

where $\bullet\{\iota'/\iota\}$ is standard type variable substitution with $\bullet$ being either a Θ, Ω, or τ. For $\iota = \overline{\mathtt{pt}};\overline{\mathtt{mt}}$ and $\iota' = \overline{\phi};\overline{\mu}$, the operator substitutes every occurrence of $\mathtt{pt}_i$ with ϕ_i and every occurrence of $\mathtt{mt}_j$ with μ_j. The partial function is defined iff $|\overline{\mathtt{pt}}| = |\overline{\phi}|$, $|\overline{\mathtt{mt}}| = |\overline{\mu}|$, $\overline{\mathtt{pt}} \cap \overline{\phi} = \emptyset$, $\overline{\mathtt{mt}} \cap \overline{\mu} = \emptyset$. Predicate $\langle\overline{\phi};\overline{\mu}\rangle \Subset \langle\overline{\phi}';\overline{\mu}'\rangle$ holds iff $\overline{\phi} \subseteq \overline{\phi}' \cup \mathtt{whole}(\overline{PO})$ and $\overline{\mu} \subseteq \overline{\mu}' \cup \mathtt{whole}(\overline{MO})$ where $\Psi = \overline{PO}\ \overline{MO}\ \overline{C}$. Following the FGJ, all definitions and rules in this paper are implicitly parameterized by the immutable code base Ψ.

Well-formedness Judgment $\Delta \vdash_{\mathtt{wft}} \tau$ says type τ is well-formed under Δ, defined in Fig. 5. Judgment $\iota \vdash_{\mathtt{wfp}} \Theta$ says phase constraint set Θ is well-formed under ι and $\iota \vdash_{\mathtt{wfm}} \Omega$ says mode constraint set Ω is well-formed under ι. Judgment $\vdash_{\mathtt{wfe}} \Delta$ says energy specification Δ is well-formed. Their definitions are defined in Fig. 6.

The most important rules are perhaps (WF-PSet) and (WF-MSet). (WF-PSet) characterizes the essence of "phase distinction": any two phases with distinct names as declared

in $\overline{PO}$ should never unify – the restriction of Θ to constant phase names is thus an identity relation. (WF-MSet) says that the ordering defined by $\overline{MO}$ must not be violated – the restriction of Ω to constant mode names indeed yields $\overline{MO}$. In addition, both rules further require that all constant names used in constraints have been declared and all type variables are bound by the current energy characterization parameters.

$$\text{(T-Var)}\ \Gamma;\Delta \vdash \mathtt{x} : \Gamma(\mathtt{x})$$

$$\text{(T-New)}\ \frac{\mathtt{fields}(\tau) = \overline{\tau}\ \overline{\mathtt{fd}} \qquad \Gamma;\Delta \vdash\ \overline{e} : \overline{\tau} \qquad \Delta \vdash_{\mathtt{wft}} \tau}{\Gamma;\Delta \vdash\ \textbf{new}\ \tau(\overline{e}) : \tau}$$

$$\text{(T-Cast)}\ \frac{\Gamma;\Delta \vdash e : \tau' \qquad \Delta \vdash_{\mathtt{wft}} \tau}{\Gamma;\Delta \vdash (\tau)e : \tau}$$

$$\text{(T-Adapt)}\ \frac{\Gamma;\Delta \vdash e : \tau \qquad \Delta = \iota;\Theta;\Omega \qquad \phi \Subset \iota}{\Gamma;\Delta \vdash \textbf{adapt}[\phi]\ e :\ (\tau \Leftarrow \phi)}$$

$$\text{(T-Mswitch)}\ \frac{\Gamma;\Delta \vdash e : \textbf{modev} \qquad \overline{\Delta'} = \Delta \Leftarrow \overline{\mathtt{m}} \qquad \Gamma;\overline{\Delta'} \vdash \overline{e} : \tau \qquad \Gamma;\Delta \vdash e' : \tau}{\Gamma;\Delta \vdash \textbf{mswitch}(e)\{\overline{\mathtt{m}} : \overline{e}, e'\} : \tau}$$

$$\text{(T-Msg)}\ \frac{\begin{array}{c}\Gamma;\Delta \vdash\ e : \tau \qquad \Gamma;\Delta \vdash \overline{e} : \overline{\tau}\{\iota_0/\iota'\} \\ \mathtt{mtype}(\mathtt{md},\tau) = \Delta'.(\overline{\tau} \to \tau') \\ \Delta \vdash_{\mathtt{inst}} \iota_0 \downarrow \Delta' \\ \Delta = \iota;\Theta;\Omega \\ \Delta' = \iota';\Theta';\Omega' \\ \mathtt{phase}(\Delta) \cong \mathtt{phase}(\tau) \in \Theta \\ \mathtt{mode}(\tau) \mathtt{<:mode}(\Delta) \in \Omega\end{array}}{\Gamma;\Delta \vdash e.\mathtt{md}\langle\iota_0\rangle(\overline{e}) : \tau'\{\iota_0/\iota'\}}$$

$$\text{(T-Field)}\ \frac{\Gamma;\Delta \vdash e : \tau \qquad \mathtt{fields}(\tau) = \overline{\tau}\ \overline{\mathtt{fd}} \qquad \Delta = \iota;\Theta;\Omega \qquad \mathtt{phase}(\Delta) \cong \mathtt{phase}(\tau) \in \Theta}{\Gamma;\Delta \vdash e.\mathtt{fd}_i : \tau_i}$$

$$\text{(T-Sub)}\ \frac{\Gamma;\Delta \vdash e : \tau \qquad \tau <: \tau'}{\Gamma;\Delta \vdash e : \tau'}$$

Figure 7. Expression Typing

Expression Typing Expression typing is defined in Fig. 7. Judgment $\Gamma;\Delta \vdash e : \tau$, which means expression e has type τ under *typing environment* Γ and energy specification Δ. Typing environment Γ is defined as sequence $\overline{\mathtt{x} \mapsto \tau}$ and $\Gamma(\mathtt{x})$ as τ_i where $\mathtt{x}_i = \mathtt{x}$ and i is the right most position in Γ where $\mathtt{x}$ occurs. Notations such as $\Gamma;\Delta \vdash \overline{e} : \overline{\tau}$ denotes $\Gamma;\Delta \vdash e_1 : \tau_1, \ldots, \Gamma;\Delta \vdash e_n : \tau_n$. We define $\mathtt{ethis}(\mathtt{c}\langle\iota\rangle) \stackrel{\text{def}}{=} \mathtt{ethis}(\iota)$ and $\mathtt{ethis}(\iota;\Sigma) \stackrel{\text{def}}{=} \mathtt{ethis}(\iota)$.

$$\text{(S-Refl)}\ \vdash \tau <: \tau$$

$$\text{(S-Trans)}\ \frac{\vdash \tau <: \tau' \qquad \vdash \tau' <: \tau''}{\vdash \tau <: \tau''}$$

$$\text{(S-Class)}\ \frac{\textbf{class}\ \mathtt{c}\ \Delta\ \textbf{extends}\ \tau \cdots \in \Psi \qquad \Delta = \iota';\Sigma}{\vdash \mathtt{c}\langle\iota\rangle <: \tau\{\iota/\iota'\}}$$

Figure 8. Subtyping

Given $\mathtt{ethis}(\iota) = \langle\phi;\mu\rangle$, we define overloaded functions `phase` and `mode` as:

$$\mathtt{phase}(\mathtt{c}\langle\iota\rangle) \stackrel{\text{def}}{=} \phi \qquad \mathtt{mode}(\mathtt{c}\langle\iota\rangle) \stackrel{\text{def}}{=} \mu$$
$$\mathtt{phase}(\iota;\Sigma) \stackrel{\text{def}}{=} \phi \qquad \mathtt{mode}(\iota;\Sigma) \stackrel{\text{def}}{=} \mu$$

(T-Adapt) and (T-Mswitch) both involve type coercion. (T-Adapt) coerces the phase of the object, whereas (T-Mswitch) coerces the mode of the context. We designed the two expressions with coercions on opposite parties (context vs. object) because we found they happened to be the most natural way of programming: the **adapt** expression is used when the programmer intends to adjust the message receiver *object* to become compatible with the phase where it is used, whereas the **mswitch** is commonly used when the mode of the current *context* is "readjusted" based on the physical battery state.

The abbreviated **adapt** e expression we used in the example in Fig. 1 can be encoded as **adapt**$[\phi]\ e$ where $\phi = \mathtt{phase}(\Delta)$ – adapting to the phase of the enclosing object. The **mswitch** expression is designed to resemble the Java-like **switch** expression to promote the common programming pattern that mode-adaptive code is often "multiplexed" based on energy states. Evaluating an expression e under a particular coerced energy context (say `hifi`) can be simply encoded as **mswitch**$(\mathtt{hifi})\{\mathtt{hifi} : e, e\}$. The object/context coercions are achieved by an overloaded $\Leftarrow$ operator. Given $\iota = \phi, \overline{\phi}; \mu, \overline{\mu}$, the operator is defined as:

$$\mathtt{c}\langle\iota\rangle \Leftarrow \phi' \stackrel{\text{def}}{=} \mathtt{c}\langle\phi', \overline{\phi}; \mu, \overline{\mu}\rangle$$
$$\iota;\ \Sigma \Leftarrow \mu' \stackrel{\text{def}}{=} \phi, \overline{\phi}; \mu', \overline{\mu};\ \Sigma$$

The rules that define how objects interact are (T-Msg) and(T-Field). The most interesting aspect of (T-Msg) is perhaps the two constraints that capture the essence of waterfall invariants. Constraint $\mathtt{phase}(\Delta) \cong \mathtt{phase}(\tau)$ says the phases of the message sender and the message receiver must unify, intuitively a manifestation of phase isolation. Constraint $\mathtt{mode}(\tau)\mathtt{<:mode}(\Delta)$ says that the receiver's mode must be a less energy consuming one than the sender's. In (T-Field), we further enforces phase isolation on public field ac-

$$\text{(T-Class)}\ \frac{\begin{array}{c}\texttt{ethis}(\Delta) = \texttt{ethis}(\tau) \\ \vdash_{\texttt{wfs}} \Delta \qquad \overline{F} = \overline{\tau}\ \overline{\texttt{fd}} \\ \Delta \vdash_{\texttt{wft}} \overline{\tau}, \tau \qquad \overline{M}\ \text{OK IN}\ \texttt{c}, \Delta, \tau \\ \texttt{constructorOK}(K, \texttt{c}, \overline{F}, \tau)\end{array}}{\textbf{class}\ \texttt{c}\ \Delta\ \textbf{extends}\ \tau\{K\ \overline{F}\ \overline{M}\}\ \text{OK}}$$

$$\text{(T-Method)}\ \frac{\begin{array}{c}\Delta' = \iota; \Sigma \qquad \Delta + \Delta' \vdash_{\texttt{wft}} \overline{\tau}, \tau \\ \overline{x} : \overline{\tau}, \textbf{this} : \texttt{c}\langle\iota\rangle; \Delta + \Delta' \vdash e : \tau \\ \texttt{override}(\texttt{md}, \tau', \Delta.(\overline{\tau} \to \tau))\end{array}}{\Delta\ \tau\ \texttt{md}(\overline{\tau}\ \overline{x})\{\ e\ \}\ \text{OK IN}\ \texttt{c}, \Delta', \tau'}$$

Figure 9. Class Typing

cess. Removing this requirement does not affect soundness, but we find the requirement helpful because a large number of cross-phase public field accesses – as exhibited by many **adapt**'s – usually signify the need for more refactoring. The two rules use two FGJ standard functions: $\texttt{mtype}(\texttt{md}, \tau)$ computes the signature for method md of object τ, in the form of $\Delta.(\overline{\tau} \to \tau)$ where Δ is the energy specification associated with the method; $\texttt{fields}(\tau)$ computes the field signatures for object τ in the form of $\overline{\tau}\ \overline{\texttt{fd}}$. These functions are delayed to the Appendix. Both instantiation-time and messaging-time polymorphism are supported. $\texttt{fields}(\tau)$ is also used for (T-New), object instantiation.

(T-Cast) models Java-style casting. We do not refine this rule with a stupid cast warning [14] in this formalization, as it does not affect the type soundness result. Note that the casting expression is only meant for the coercion of Java nominal types, not the coercion of phases or modes. The latter goals should be achieved either via **adapt** and **mswitch** respectively. Our dynamic semantics (the dynamic check inserted for casting) will ensure that any abuse of the casting expression for mode/phase coercion would fail. Indeed, the $(\tau)e$ syntax we use here is only meant for facilitating the formalization; in the implemented language, the programmer syntax for casting is $(\texttt{c})e$. (T-Var) models variable typing. (T-Sub) is used for subtyping. Subtyping relation $\tau <: \tau'$ is defined in Fig. 8, including the predictable rules for reflexivity (S-Refl), transitivity (S-Trans), and Java-style nominal typing (S-Class).

Program Typechecking A program $\Psi = \overline{PO}\ \overline{MO}\ \overline{C}$ typechecks, denoted as $\vdash_{\texttt{P}} \Psi$, iff $\overline{PO}$ is a partial order, $\overline{MO}$ is a lattice, and C OK for each C in $\overline{C}$. The last part, class typing, is defined in Fig. 9. Condition $\texttt{ethis}(\Delta) = \texttt{ethis}(\tau)$ guarantees that in the presence of inheritance, the superclass and the subclass assume the same phase and mode for objects instantiated from the subclass. The binary $+$ operator over energy specifications is defined as:

$$\frac{\Delta_i = \overline{\texttt{pt}}_i; \overline{\texttt{mt}}_i; \Theta_i; \Omega_i, i = 1, 2 \qquad \overline{\texttt{pt}}_1 \cap \overline{\texttt{pt}}_2 = \emptyset \qquad \overline{\texttt{mt}}_1 \cap \overline{\texttt{mt}}_2 = \emptyset}{\Delta_1 + \Delta_2 \stackrel{\text{def}}{=} (\overline{\texttt{pt}}_1, \overline{\texttt{pt}}_2); (\overline{\texttt{mt}}_1, \overline{\texttt{mt}}_2); (\Theta_1 \cup \Theta_2); (\Omega_1 \cup \Omega_2)}$$

FGJ-like predicate $\texttt{constructorOK}(K, \texttt{c}, \overline{\tau}\ \overline{\texttt{fd}}, \tau)$ holds iff $K = \texttt{c}(\overline{\tau}'\ \overline{\texttt{fd}}', \overline{\tau}\ \overline{\texttt{fd}})\{\textbf{super}(\overline{\texttt{fd}}'); \textbf{this}.\overline{\texttt{fd}} := \overline{\texttt{fd}}\}$ and $\texttt{fields}(\tau) = \overline{\tau}'\ \overline{\texttt{fd}}'$. Predicate $\texttt{override}(\texttt{md}, \tau', \Delta.(\overline{\tau} \to \tau))$ holds iff function $\texttt{mtype}(\texttt{md}, \tau')$ is either undefined, or it is computed as $\Delta.(\overline{\tau} \to \tau)$.

Additional ET Type System Features The type system implemented by ET supports two additional features:

- *polymorphic type inference*: all type variables in the energy characterization parameters (ι) and all type constraints (Σ) are fully inferred.
- *mode subtyping*: we allow τ to be a subtype of τ' if τ and τ' are identical except $\texttt{mode}(\tau) <: \texttt{mode}(\tau')$. Intuitively, this aligns with our intuition that in energy management, implicitly replacing a program fragment with a potentially less energy-consuming one is OK, but not *vice versa*.

The accompanying technical report [37] formalizes these features, and the enriched type system is also proved sound. The type inference algorithm is a direct adaptation of the presented system in which the smallest number of constraints for Θ and Ω that satisfy the typechecking algorithm are collected. The challenging issue of determining the size of ι is addressed through one key insight: the labels used to differentiate contexts in context-sensitive algorithms [25] serve as an ideal naming scheme for type variable refreshing/instantiation. Flexible mode subtyping requires variance on parametric types. This is a thoroughly explored topic (*e.g.* [15]); ET's type inference determines variance by use and requires no annotations from programmers.

3.3 Dynamic Semantics

Relation $e \stackrel{\Xi}{\Longrightarrow} e'$ denotes expression e is one-step reduced to e' under context object characterization Ξ. Runtime expressions and values are defined in Fig. 10.

$$\begin{array}{lll}
e ::= \cdots \mid \textbf{cl}(\Xi, e, e') \mid\ e; e' \mid \!\!\rightarrow \Xi \mid v & & \textit{runtime expressions} \\
v ::= o \mid \texttt{m} & & \textit{value} \\
o ::= \textbf{obj}(\tau, \tau', \overline{v}) & & \textit{object value} \\
\mathbf{E} ::= \odot \mid\ \textbf{new}\ \tau(\ldots, v, \mathbf{E}, e, \ldots) & & \textit{evaluation context} \\
\quad \mid\ \textbf{cl}(\Xi, \mathbf{E}, e) & & \\
\quad \mid\ \mathbf{E}.\texttt{md}\langle\iota\rangle(\overline{e}) & & \\
\quad \mid\ o.\texttt{md}\langle\iota\rangle(\ldots, v, \mathbf{E}, e, \ldots) & & \\
\quad \mid\ (\tau)\mathbf{E} \mid\ \textbf{adapt}[\texttt{p}]\ \mathbf{E} & & \\
\quad \mid\ \textbf{mswitch}(\mathbf{E})\{\overline{\texttt{m} : e}, e\} & &
\end{array}$$

Figure 10. Run-Time Elements

Expression $\mathbf{cl}(\Xi, e, e')$ is a function closure, *i.e.* expression e under context object characterization Ξ, with destructor e' evaluated before the closure is destructed. $e; e'$ is a standard expression continuation. Expression $\looparrowright \Xi$ indicates that CPU should be scaled to the frequency associated with context Ξ. Values can either be a mode value, or an object value $\mathbf{obj}(\tau, \tau', \overline{v})$ where τ is the type of the object at its instantiation, τ' is its current type, and $\overline{v}$ holds values of the fields.

Notation $\langle \Xi; e \rangle \Uparrow$ denotes that computation e under Ξ diverges. A reduction is stuck if a pre-condition is not satisfied. The initial configuration of reduction for program Ψ, denoted as $\texttt{init}(\Psi)$, is $\langle \Delta_{\texttt{main}}; e \rangle$ where $\Delta_{\texttt{main}} = \texttt{p}_{\texttt{main}}; \texttt{m}_{\texttt{main}}; \epsilon; \epsilon$ and $\texttt{p}_{\texttt{main}}$ and $\texttt{m}_{\texttt{main}}$ are built-in phase/-mode names for the bootstrapping code. In addition, we have $\texttt{mbody}(\texttt{main}\langle\epsilon\rangle, \texttt{Main}\langle\epsilon\rangle) = \epsilon.e$ under code base Ψ.

The reduction rules are defined in Fig. 11. (R-Msg) and (R-MsgScale) defines the behavior for messaging. Scaling is needed when the message sender and message receiver do not agree on the phases. In both cases, $\texttt{ethis}(\tau)$ keeps the "original phase" of the object o, *i.e.* the phase associated with its instantiation point. The decision of scaling is made by comparing it with the phase of the context. If the two "agree" – such as the two are declared with the same phase (the definition of agreement will be given shortly) – no scaling is needed. The FGJ standard function $\texttt{mbody}(\texttt{md}\langle\iota\rangle, \tau)$ computes the method body of method $\texttt{md}$ of object τ with its type parameters instantiated with ι, in the form of $\overline{\texttt{x}}.e$ where $\overline{\texttt{x}}$ are arguments that may occur free in body e. The phase agreement relation is defined as:

$$\langle \texttt{p}; \texttt{m} \rangle \sim \langle \texttt{p}'; \texttt{m}' \rangle \quad \stackrel{\text{def}}{=} \quad \mathcal{SM}(\texttt{p}) = \mathcal{SM}(\texttt{p}')$$

where $\mathcal{SM} : \mathsf{PCONST} \rightarrow \mathsf{AF}$ is a monotone function and AF represents a total order of available frequencies or frequency/voltage pairs. Intuitively, $\mathcal{SM}$ maps each distinct programmer-declared phase name to an available frequency (in AF), which is specific to the hardware and OS. There is no need to scale iff the message sender's phase and the receiver's phase map to the same frequency.

Since our reduction system does not explicitly model CPU state, the scaling operator itself is defined as a no-op, as in (R-Scale). The important issue of scaling factor selection however should not be ignored. Indeed, the short answer would be that the CPU should scale to frequency $\mathcal{SM}(\texttt{p}')$ given $\Xi' = \langle \texttt{p}'; \texttt{m}' \rangle$. When the lattice of mode ordering happens to fall into a total order, we in addition support *proportional scaling*, a case of interaction between phases and modes. Here, we proportionally set the range of CPU frequencies program phases can map to *based on mode information*. Under a "lower" mode, the range of CPU frequencies to be assigned to phases should not include the highest ones – when system energy state is low, the program should save as much energy as possible by not running code on the most expensive frequencies. Formally, this means that the frequency should be used is $\mathcal{SM}_{\texttt{m}}(\texttt{p})$, where mapping $\mathcal{SM}_{\texttt{m}} : \mathsf{PCONST} \rightarrow \mathsf{AF}_{\texttt{m}}$ is a monotone function s. t. $\mathsf{AF}_{\texttt{m}} \subseteq \mathsf{AF}$, and for any $\texttt{m}_1 \texttt{<:} \texttt{m}_2 \in (\overline{MO})^*$, $\ulcorner \mathsf{AF}_{\texttt{m}_1} \urcorner \leq \ulcorner \mathsf{AF}_{\texttt{m}_2} \urcorner$ and $\llcorner \mathsf{AF}_{\texttt{m}_1} \lrcorner \leq \llcorner \mathsf{AF}_{\texttt{m}_2} \lrcorner$, where $\ulcorner \urcorner$ and $\llcorner \lrcorner$ compute the greatest and smallest elements of the total order respectively and $\Psi = \overline{PO}\ \overline{MO}\ \overline{C}$.

(R-MSwitchM) and (R-MSwitchD) define the behavior of the **mswitch** expression. If there is a match for the mode value, (R-MSwitchM) applies and note that the mode of the context object characterization is coerced. Otherwise, the default behavior applies according to (R-MSwitchD). (R-Adapt) coerces the phase of the object. (R-New), (R-Field),(R-Cast), (R-Closure) define the standard behaviors of object instantiation, field access, casting, and closure elimination. (R-Context) reduces the redex in the evaluation context, whose definition appears in Fig. 10. We use notation $\mathbf{E}_\Xi$ to represent the evaluation context with the $\mathbf{cl}(\Xi, e, e')$ expression immediately enclosing the hole ($\odot$). Formally, $\mathbf{E}_\Xi[]$ is defined as $\mathbf{cl}(\Xi, \mathbf{E}[], e)$.

It should be noted that the reduction rules here – and the data structures they operate on – are optimized for proofs, not for implementation. For instance, we choose to have the function closure $\mathbf{cl}(\Xi, e, e')$ to dynamically carry the context object characterization Ξ, and the heap object $\mathbf{obj}(\tau, \tau', \overline{v})$ to dynamically carry the coerced type τ'. As a result, (R-MswitchM), (R-Adapt), and (R-New) all need to dynamically maintain these data structures. In fact, both pieces of dynamically carried information can be statically inferred and hence safely erased from the dynamic semantics. On the high level, the erasure of Ξ from the construct of function closure also says that the decision of scaling – the choice between (R-Msg) and (R-MsgScale) – can be statically determined, a fact that comes with no surprise considering our static approach. Since we are not presenting the inference algorithm, we do not formally define the erasure semantics here. Our implemented language does support full inference, and hence the erasure of Ξ from $\mathbf{cl}(\Xi, e, e')$ and τ' from $\mathbf{obj}(\tau, \tau', \overline{v})$.

3.4 Properties

In this section, we describe several properties of ET. All proofs can be found in a technical report [37]. The typing rules for the auxiliary expressions are given in Fig. 12.

Properties of the Type System The main property is soundness, which is proved via subject reduction and progress:

Lemma 1 (Subject Reduction)**.** *If* $\Gamma; \Delta \vdash e : \tau$*,* $e \stackrel{\texttt{ethis}(\Delta)}{\Longrightarrow} e'$*, then* $\Gamma; \Delta \vdash e' : \tau$*.*

Also known as the type preservation lemma, subject reduction – from the angle of energy-aware programming – guarantees a computation, *i.e.* expressions over a reduction sequence, has to commit to one phase and one mode, a computational interpretation of *stable* energy consumption traits of program elements.

(R-Msg)	$o.\mathtt{md}\langle\iota\rangle(\overline{v}') \xrightarrow{\Xi} \mathbf{cl}(\Xi', e\{\overline{v}'/\overline{\mathtt{x}}\}\{o/\mathbf{this}\}, v)$	if $\Xi' = \mathtt{ethis}(\tau), \Xi \sim \Xi'$, any v
(R-MsgScale)	$o.\mathtt{md}\langle\iota\rangle(\overline{v}') \xrightarrow{\Xi} \looparrowright \Xi'; \mathbf{cl}(\Xi', e\{\overline{v}'/\overline{\mathtt{x}}\}\{o/\mathbf{this}\}, \looparrowright \Xi)$	if $\Xi' = \mathtt{ethis}(\tau), \Xi \sim \Xi'$ false
(R-Scale)	$\looparrowright \Xi'; e \xrightarrow{\Xi} e$	
(R-MswitchM)	$\mathbf{mswitch}\ (\mathtt{m}_i)\ \{\overline{\mathtt{m}:e}, e_0\} \xrightarrow{\Xi} \mathbf{cl}(\langle \mathtt{p}; \mathtt{m}_i\rangle, e_i, v)$	any v
(R-MswitchD)	$\mathbf{mswitch}\ (\mathtt{m}')\ \{\overline{\mathtt{m}:e}, e_0\} \xrightarrow{\Xi} e_0$	if $\mathtt{m}' \neq \mathtt{m}_i$
(R-Adapt)	$\mathbf{adapt}[\mathtt{p}']o \xrightarrow{\Xi} \mathbf{obj}(\tau, \tau' \Leftarrow \mathtt{p}', \overline{v})$	
(R-New)	$\mathbf{new}\ \tau_0(\overline{v}') \xrightarrow{\Xi} \mathbf{obj}(\tau_0, \tau_0, \overline{v}')$	
(R-Field)	$o.\mathtt{fd}_i \xrightarrow{\Xi} v_i$	
(R-Cast)	$(\tau_0)o \xrightarrow{\Xi} o$	if $\tau' <: \tau_0$
(R-Closure)	$\mathbf{cl}(\Xi', v, e_0) \xrightarrow{\Xi} e_0; v$	
(R-Context)	$\mathbf{E}_{\Xi'}[\,e_1\,] \xrightarrow{\Xi} \mathbf{E}_{\Xi'}[\,e_2\,]$	if $e_1 \xrightarrow{\Xi'} e_2$

for all rules: $o = \mathbf{obj}(\tau, \tau', \overline{v})$, $\mathtt{mbody}(\mathtt{md}\langle\iota\rangle, \tau') = \overline{\mathtt{x}}.e$, and $\Xi = \langle \mathtt{p}; \mathtt{m}\rangle$

Figure 11. Reduction Rules

$$\text{(T-Cl)}\ \frac{\Gamma;\Delta' \vdash e : \tau \qquad \mathtt{ethis}(\Delta') = \Xi \qquad \Gamma;\Delta \vdash e' : \tau'}{\Gamma;\Delta \vdash\ \mathbf{cl}(\Xi, e, e') : \tau}$$

$$\text{(T-Scale)}\ \frac{\text{any } \tau}{\Gamma;\Delta \vdash \looparrowright \Xi : \tau}$$

$$\text{(T-Mv)}\ \frac{\mathtt{m} \in \mathtt{whole}(\overline{MO}) \qquad \Psi = \overline{PO}\ \overline{MO}\ \overline{C}}{\Gamma;\Delta \vdash \mathtt{m} : \mathbf{modev}}$$

$$\text{(T-Obj)}\ \frac{\Gamma;\Delta \vdash \overline{v} : \overline{\tau} \qquad \tau' = \tau \Leftarrow \phi \text{ for some } \phi \qquad \mathtt{fields}(\tau) = \overline{\tau}\ \overline{\mathtt{fd}}}{\Gamma;\Delta \vdash\ \mathbf{obj}(\tau, \tau', \overline{v}) : \tau'}$$

$$\text{(T-Cont)}\ \frac{\Gamma;\Delta \vdash e : \tau' \qquad \Gamma;\Delta \vdash e' : \tau}{\Gamma;\Delta \vdash\ e; e' : \tau}$$

Figure 12. Auxiliary Run-time Expression Typing

Definition 1 (Bad Cast)**.** *Expression* $(\tau_0)\mathbf{obj}(\tau, \tau', \overline{v})$ *is a bad cast iff* $\tau' <: \tau_0$ *does not hold.*

Lemma 2 (Progress)**.** *If* $\Gamma;\Delta \vdash e : \tau$*, then either* $e \in \mathsf{V}$*, or* e *is a bad cast, or there exists some* e' *such that* $e \xrightarrow{\mathtt{ethis}(\Delta)} e'$*.*

Theorem 1 (Type Soundness)**.** *If* $\Gamma;\Delta \vdash e : \tau$*, then either* $e \xrightarrow{\Xi}_* v$*, or* $\langle \Xi; e\rangle \Uparrow$*, or* e *is a bad cast, where* $\Xi = \mathtt{ethis}(\Delta)$*.*

The theorem here does not explicitly relate a statically typed program Ψ with the run time, which we state now as Thm.2:

Theorem 2 (Sound Static Typing)**.** *If* $\vdash_{\mathtt{p}} \Psi$*,* $\mathtt{init}(\Psi) = \langle \Xi; e\rangle$*, then* $e \xrightarrow{\Xi}_* v$*, or* $\langle \Xi; e\rangle \Uparrow$*, or* e *is a bad cast.*

Last we state decidable type checking:

Theorem 3 (Type Decidability)**.** *For any program* Ψ*, it is decidable whether* $\vdash_{\mathtt{p}} \Psi$ *holds.*

Mode and Phase Invariants We now prove that both phase isolation and the waterfall invariant for modes hold.

Definition 2 (Redex)**.** $\langle \Xi; e\rangle$ *is a redex over program* Ψ *iff* $e_0 \xrightarrow{\mathtt{ethis}(\Delta_0)}_* \mathbf{E}_\Xi[e]$ *where* $\mathtt{init}(\Psi) = \langle \Delta_0; e_0\rangle$*.*

Theorem 4 (Phase Isolation)**.** *Given a program* Ψ *such that* $\vdash_{\mathtt{p}} \Psi$ *and any redex* $\langle \mathtt{p}; \mathtt{m}; \mathbf{obj}(\tau, \tau', \overline{v}').\mathtt{md}\langle\iota\rangle(\overline{v})\rangle$*, then* $\mathtt{phase}(\tau') = \mathtt{p}$*.*

Observe here that τ' in $\mathbf{obj}(\tau, \tau', \overline{v}')$ is the "current type" in the presence of adaptation, so this property says that either 1) there is no adaptation ($\tau = \tau'$) and then the phase of the object at instantiation time and the phase of the calling object must unify, or 2) there is adaptation ($\tau \neq \tau'$), and then the phase of the object after the last adaptation and the phase of the calling object must unify.

Theorem 5 (Waterfall Invariant)**.** *Given a program* Ψ *such that* $\vdash_{\mathtt{p}} \Psi$ *and any redex* $\langle \mathtt{p}; \mathtt{m}; \mathbf{obj}(\tau, \tau', \overline{v}').\mathtt{md}\langle\iota\rangle(\overline{v})\rangle$*,* $\mathtt{mode}(\tau) <: \mathtt{m} \in \overline{MO}$ *where* $\Psi = \overline{PO}\ \overline{MO}\ \overline{C}$*.*

Note that the **mswitch** expression directly updates the context object characterization, so this lemma subsumes **mswitch** coercion.

4. Implementation and Evaluation

This section describes an ET compiler implementation, some programming case studies and benchmarking results of ET programs on the Android operating system [1].

4.1 Compiler

ET has been implemented on top of the Polyglot compiler framework [28]. All features presented in the formal system have been implemented. The compiler in addition implemented constraint-based type inference, and some convenience features such as class-level phase/mode qualifiers, and a `noscale` declaration for short methods (such as getters and setters) where the otherwise small overhead of frequency scaling may outweigh the benefit. The source code of the compiler can be downloaded online [37].

The solutions of the phase type variables in type inference are mapped back to the AST, so that the pass of code generation can decide whether DVFS calls need to be inserted (*i.e.* type-directed frequency scaling). The code for frequency scaling is written in C, and interfaces with target code through Java Native Interface (JNI). A small portion of target code specific to Android, such as setting governors, is also automatically generated.

Our compiler can analyze Android library sources as ET programs. To avoid analyzing uninteresting library code which artificially increases LOC count, our compiler conservatively excludes library code that does not affect a parametric analysis, such as method invocations with primitive data as arguments and return values, or operations on built-in Java objects such as `System` and `String`.

4.2 Evaluation

We have converted several Android programs into ET, with basic information as follows. `SpaceBlaster` and `Asteroids` are from a reference book [33]; the rest are from Android's developer site [1]. The selection is intended to cover Android Apps of distinct natures, such as games, GUI applications, I/Os, and mathematical computations.

name	description	LOC
`Asteroids`	interactive game	2236
`BackupRestore`	file I/O	326
`CubeLiveWallpaper`	GUI design	241
`GesturesDemo`	gesture recognition	3205
`SpaceBlaster`	interactive game	1700

Programming Experience We found that the amount of effort required to convert a Java program to an ET program was mild. Incremental programming, *i.e.* starting from a Java program and adding ET features gradually, does not appear difficult in our case studies. We now use `SpaceBlaster` as an example to describe our programming experience in greater detail.

The `SpaceBlaster` program is declared with 4 phases, the `main` phase, the `math` phase for the computation-intensive part of the program, the `graphics` phase for rendering-related operations, and the `audio` phase for sound-related operations. Their impact on DVFS is reflected in our phase definition:

```
phases {
        main <cpu math;
        graphics <cpu main;
        audio <cpu main;
}
```

The modes declared in `SpaceBlaster` are `lofi`, `hifi`, and `full`, in a similar way as in the example we used in Sec. 2. The program contains 17 **adapt** and 8 **mswitch** expressions, a relatively small programming effort for a program with 1700LOC. The attribution of most objects to phases is fairly predictable. For modes, each rendering element (the `BitMap` objects) – the ship, the meteor, the fires, and the bullet – is defined with two or three different implementations, each using a slightly different pixel size. We found that several operations, specifically some rendering operations, have a high impact on energy consumption because games have a relatively high frame rate for graphics rendering. We use **mswitch** in these operations to select a smaller or larger bitmap, depending on the current battery state.

We now report the insights we gained and the lessons we learned throughout this experience.

First, it is common in Android programming to unnecessarily intertwine operations of distinct CPU usage patterns (see Benefit 2). The original `SpaceBlaster` code mixed physics-related mathematical computations – such as computing the background, computing the velocity of the spaceship – with their display. After declaring `graphics` and `math` as separate phases, phase isolation naturally guided us in separating physics-related operations into an object of its own, which is only **adapt**'ed once from the `graphics` phase to compute physics for each game frame.

Second, for objects that genuinely cross phase boundaries – typically objects recording game data – the interesting question is which phases they should be declared with. As lazy programmers, we naturally wish to use as few **adapt**'s as possible. The decision of selecting an appropriate phase for declaration is thus based on how often or complex such objects are used in individual phases. For instance, if the uses of the `Position` object are scattered through the `math` phase, but are very few and concentrated in the `graphics`, it is our inclination to declare it as a `math` object and only **adapt** when being used in `graphics`. We speculate this methodology also improves data locality, which, with advanced data memory management, further contributes to energy efficiency (see Sec. 6).

Third, Benefit 4 is effective in helping us organize programs. Among the compiler error messages given by ET, we find the messages involving the violation of the waterfall invariant very useful for improving the code quality. For in-

stance, we have a resource container `ResContainer` class that keeps the ship, meteor, bullet, *etc*, instances:

```
class ResContainer {
    SBBitmap ship;
    SBBitmap bullet;
    void setShip(SBBitmap s) {
        ship = s;
    }
    void setBullet(SBBitmap b) {
        bullet = b;
    }
    void setFM(int i) {
        ship.setFM(i);
        bullet.setFM(i);
    }
    ...
}
```

In a multi-mode program such as ours, it is natural to associate each mode with a `ResContainer` object. We need a methodology that helps prevent, for instance, a `ResContainer` object of `lofi`mode from accidently including a ship object which is `hifi`. In the following code, it would be a type-error if the last line were included:

```
SBBitmap@mode(hifi) ship_high =
        MyBitmapFactory.load(...);
SBBitmap@mode(lofi) ship_low =
        MyBitmapFactory.load(...);
...
res_high = new ResContainer@mode(hifi)();
res_low = new ResContainer@mode(lofi)();
res_high.setShip(ship_high);
res_low.setShip(ship_low);
res_low.setShip(ship_high);    //error
```

Fourth, the process of attributing objects to phases and modes also promotes object-oriented programming in general. The original `SpaceBlaster` program was written in a "C-style" disguised in Java syntax, with very few (yet large) classes. When we converted this application to ET, and were required to add phase and mode qualifiers to our objects, the same thought process of attributing different methods and fields to different phases and modes also helps us refactor the code into more objects.

Fifth, the type coercion aspect of **adapt** and **mswitch** is useful to maintain the flexibility of the type system. One of the commonly cited problems of a type system approach is its innate conservativeness may disallow useful – but exceptional to the type invariant – programs. For instance, the waterfall invariant dictates that an object in the `lofi` mode should not send messages to a `hifi` object directly. In our case studies, we occasionally run into a situation where an Android library object, say `libHi`, is used mostly in the high battery state – hence justifiably declared as `hifi` – but one or two of its methods, say `lowUse`, may also need to be invoked in the `lofi` mode. This problem can typically be avoided by refactoring – such as breaking the class of the aforementioned object into two – but in the presence of library classes, such a solution may lead to problems of backward compatibility. With **mswitch**, the following code can appear in a `lofi` context, reminiscent of declassification in language-based security [41] or approximation endorsement in EnerJ [30]:

```
mswitch(hifi) {hifi: libHi.lowUse();}
```

The programming experience also revealed one possible weakness of Energy Types. Because we do not have a "top phase", every "utility" object has to be placed into a phase of our declaration as well. This is not a problem for those utility objects only used in one phase – our type inference algorithm will just unify them to that phase – but it involves awkward **adapt** coercions in cases such as when an object factory is used in multiple phases. To add a "top phase" as a top type is a trivial type system addition. The open question is whether full-fledged support of phase subtyping – the more elegant solution from a type theory perspective – is really beneficial to real-world programming (*i.e.* not an over-design).

Experiment Setup The target code is benchmarked on a Motorola Droid X with Android version 2.2.1 (Froyo) and API level 8. For this configuration, four CPU frequencies are available: 1Ghz, 800Mhz, 600Mhz, 300Mhz. Energy consumption is measured by Watts up? (PRO/ES/.Net) [1]. All experiments are conducted at full battery to minimize the effect of battery charging. Each application is tested under 6 scenarios:

- A is "original Android," *i.e.* Android code with no phases and modes, running at 1GHz on a governor with no frequency scaling;
- B is "phase only + full-range frequencies," *i.e.* code with phase declarations and phases are mapped to the whole range of frequency levels (300Mhz - 1Ghz);
- C is "phase only + partial-range frequencies," *i.e.* code with phase declarations and phases are mapped to a lower range of frequency levels (300Mhz - 800hz). This case is designed to test "proportional scaling," the idea we described in Sec. 3.3;
- D is "phase + degraded mode." The choice here is application-specific (see the description later). The "degraded mode" here is `lofi` and the original mode for A, B, C is `hifi`;
- E is "Android with on-demand CPU scaling." Android is equipped with an `ondemand` governor that allows for frequency scaling based on performance metrics.

[1] `www.wattsupmeters.com`

- F is "lowest frequency only," *i.e.* Android code with no phases and modes, running at 300Mhz on a governor with no frequency scaling.

Since the same program may execute in different speeds under different scenarios, the same logical task – say playing 2500 frames of a game loop – may not take the same time to complete. With energy as a cumulative value over time, the only fair energy comparison across scenarios would be to have all energy measurements conducted for the same logical task. For `Asteroids` and `SpaceBlaster`, the logical task chosen for measurement is 2500 frames of game loop; for `BackupRestore`, it is 1M integer writes; for `CubeLiveWallPaper`, it is 1250 frames of display; for `GestureDemo`, it is performing and recognizing 7 gestures.

We have modified `Asteroids` and `SpaceBlaster`, so that the two games can be run in the auto-pilot mode. The only application that involves significant user interaction is `GestureDemo`. In this case, the testing users are trained to follow the same routine (the rate of touch, the path of finger movement) until energy consumption stabilizes (usually 2-5 trials), and the mean of the next 5 trials is chosen. We found such experiments yield very stable results, with variations ranging ± 2 joules in most cases.

DVFS Overhead DVFS is known to incur a small overhead. The conventional wisdom is that the switching time is usually in the tens of microseconds (*e.g.* 20 μs for Intel Xscale) that are often ignored on application-level studies (*e.g.* [30]) and the switching energy cost is so small that even many architecture-level studies (*e.g.* [23, 39]) do not consider it. Our experiments however showed a disproportionate amount of time and energy consumed on DVFS:

- Average DVFS time: 0.95ms - 1.05ms
- Energy consumption per DVFS: 0.8 mJ - 1.0mJ.

This data is almost 1-2 magnitude higher than the assumptions of the state-of-the-art energy simulators. We believe this results from the inefficiency of JNI use for DVFS calls. To perform DVFS, we have to invoke a JNI method first, and then invoke the DVFS system calls on the C side (invoking system calls directly from the Java side is even slower). If a technology like ours turns out to be useful for Android platforms, we expect more efficient ways to perform DVFS would be developed in future Android API's. In the following report, we adjust our data in the following conservative manner:

- Adjusted DVFS time: 200μs. This is still one magnitude higher than Intel Xscale, but we would rather err on the safe side.
- Adjusted DVFS energy consumption: 0.9mJ $\times \frac{200\mu s}{1ms}$. In other words, we proportionally adjust energy consumption by time. We think this is reasonable because 1) the relation between energy consumption and time is theoretically linear; 2) considering most simulation-based energy research ignores DVFS energy cost altogether, any good-faith adjustment is indeed on the conservative side.

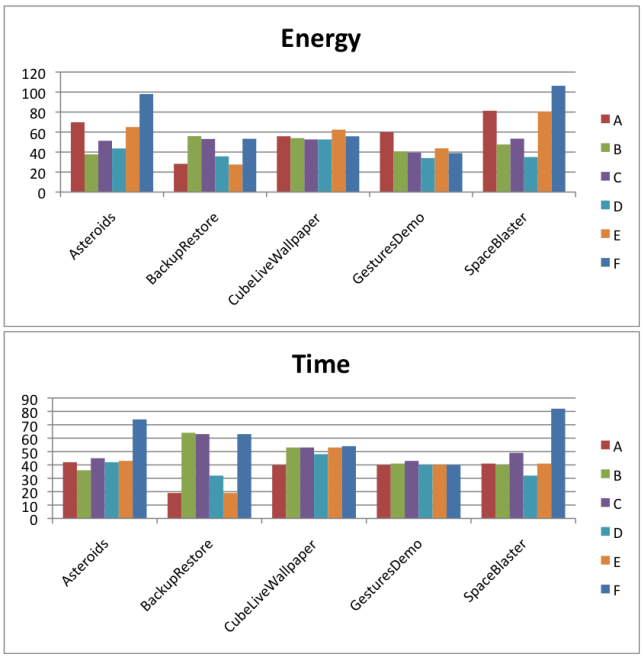

Figure 13. Benchmark Results: Energy Consumption (joules) and Time (seconds)

Benchmarking Results The benchmarking result is illustrated in Fig. 13.

Phases appear to be an effective approach for reducing energy consumption. For all experiments we conducted – with the only exception of `BackupRestore`, a case we will discuss in separation at the end of this section – energy consumption in B, C, D columns is lower than A. Even with a conservative approximation of the DVFS overhead, our experiments suggest that the gain well offsets the overhead in most cases. For `Asteroids`, `GestureDemo`, and `SpaceeBlaster`, energy consumption is reduced from A to B by 30%-50%. What is even more striking is that in all three cases, the time between A and B almost remains the same. This confirms that slowing down the CPU frequency at the graphics-intensive phase is a good choice. Note that `Asteroids` even had a mild speed up from A to B. We speculate this shows that the dominating factor in graphics-related code is the status of I/O, not CPU frequency at all.

`CubeLiveWallPaper` shows very little gain in energy savings by performing DVFS. The execution time in fact increases slightly as a result, a fact not too surprising since energy and time are often trade-offs. This benchmark – displaying some animated wall papers – maintains a very low level of computation, and we think this is a reminder that the relative effectiveness of our approach increases when the

application grows more complex. As the application grows, the "weight" of the computation makes CPU scaling "worth it."

We are relatively conservative on our experiments related to modes, because the effectiveness of this feature is application-specific. Lacking an objective framework to define, say, what a "3% fidelity degradation" means, we choose to only change certain parameters of the original Android applications so that the programs execute with no or little perceptive difference. For example, both `SpaceBlaster` and `Asteroids` reduce the image size (spaceships, asteroids) by a small percentage (such as 20 points) in the `lofi` mode, whereas `GesturesDemo` increases the stroke tolerance from 3 to 6. This is why our results (column D) do not often yield "dramatic" energy savings. Green [4] has interesting proposals on defining QoS and hence more sophisticated experiments in this regard.

What is somewhat striking is our static approach can in fact fare better than the purely dynamic approach toward CPU scaling (column E) in most cases. As we mentioned in Sec. 2, the relationship between the static approach and the dynamic approach is complementary. That being said, the experimental results suggest a standalone static approach may have a future of its own in many scenarios where programmers are willing to participate in the process of energy-efficient computing.

The most baffling case is `BackupRestore`. In this program, we have scaled down the frequency when I/O happens, which in this case means writing a large number of integers to a file. Our hope had been to take advantage of the CPU slacks resulted from file I/O, but this does not seem to work. We do not fully understand why a file I/O-intensive application is so resistant to DVFS: running the program to 30% of the original frequency increases the execution time by 3 times. The only discovery we have is that the file I/O uses `java.io.RandomAccessFile`'s `write` method, which is a C native method invoked through JNI. Considering JNI caused significant slow-downs on DVFS system calls themselves, we speculate there might be some internal mechanism with JNI that is rather CPU-intensive. The lessons we draw from this anomaly are twofold. First, there are cases whose energy consumption patterns a well-intentioned programmer simply cannot characterize correctly. A more complete approach would be a unified one involving both statics and dynamics, a blueprint we sketched at the end of Sec. 2. Second, for energy-conscious platforms such as Android, elements of the platform that have high performance and energy impacts, such as JNI, should be well documented by Android developers, as well as targets for improvement.

5. Related Work

Designing programming models for energy efficiency is a relatively nascent direction. Sensor network language Eon [34] supports data flows conditioned by programmer-defined energy states. The Green framework [4] allows programmers to specify and calibrate QoS. Flicker [21] programmers can identify non-critical data in their program so that they can be placed in a memory area with lower memory refresh rate. None of these efforts is equipped with a reasoning framework. More distant are efforts on embedded systems programming – sensor network programming in particular – that often indirectly address energy consumption by reducing execution sequence length and memory footprint. Flask [24] uses a flavor of meta programming to allow more efficient (and more energy-friendly) code. On the philosophical level, this project is aligned with a number of application-level power management approaches [9, 22] in that the spirit of "users/programmers know more" is shared.

In EnerJ [30], programmers can annotate whether data are part of an approximate computation or a precise one. For approximate data, rich hardware support is offered to save energy, such as cache-line-based object layout, width reduction for floating point operations, and reducing DRAM refresh rate. EnerJ is equipped with a type system to enforce interactions of approximate and precise data, on the high-level can be related to a 2-mode Energy Types system. EnerJ adopts an information flow model – an approximate value can neither directly nor indirectly interfere with a precise value. EnerJ does not overlap with the design of phases in ET. Recently, more refined architectural support for EnerJ-like approximate programming is developed [10].

Several formal frameworks are not directly related to energy consumption, but have the potential to impact the practice of energy-aware programming. Amortized resource analysis [12] offers precise cost-based worst-case analysis for resource usage in a functional setting. Recently, a relational assertion logic [6] is constructed to reason about program relaxation and acceptability. It provides a rigorous setting to help programmers express and verify the properties of program approximation.

Energy efficiency is more discussed in the context of compiler optimization. Kandemir *et. al.* [18] studied the impact of various compiler techniques (such as loop unrolling) on energy consumption. Fen *et. al.* [40] investigated the pros and cons of compiler-assisted dynamic voltage scaling. Hsu and Kremer [13] designed an algorithm combining profiling and static analysis to identify scaling points.

Dynamic approaches such as via online monitoring and offline profiling are the majority of today's power management techniques; for a survey, see [19].

6. Conclusion

This paper is a systematic study on constructing – and more importantly – reasoning about energy-aware software. It lays a type-theoretic foundation for phase-based and mode-based energy management, and validates the ideas in the concrete context of smartphone programming.

This paper is a small step toward a direction that calls for more investigation. Considering the advanced status of type system research, many interesting ideas beyond the scope of this paper – effect types, type states, dynamic/soft/gradual typing, cost-based resource typing, dependent types, to name a few – may be applicable for reasoning energy-aware software. Along this line, Energy Types is neither the most precise in characterizing energy consumption, nor the most expressive in capturing the intentions of energy-aware programmers. Now that the bridge between type systems and phase-based and mode-based energy management is built, the real excitement lies upon how many type system ideas are relevant in building energy-aware software. On this long wish list, an immediate future work of ET is to promote more expressive interactions between mode values and object types in a hybrid setting that involves both static reasoning and lightweight dynamic reasoning.

We are also interested in exploring how phase information can guide memory management towards energy-efficient computing. It is a widely accepted fact that cache misses and the resulting memory roundtrips are a major contributor of energy consumption [35]. So, in addition to the benefits we described in Sec. 1, phases as types may further promote energy efficiency by guiding memory systems to reduce cache misses. We are in particular interested in modifying the virtual machines to co-allocate objects in the same phase.

This paper uses DVFS as a low-level energy management strategy to demonstrate the usefulness of Energy Types. A broader question is whether and how the solutions from the lower layers of the compute stack can benefit from high-level program information such as phases and modes. For instance, workload characterization is important in virtualization [5] and energy-efficient cloud computing. It is an open question whether phases and modes can offer useful hints that scheduling and consolidation algorithms can benefit from.

Acknowledgments We thank the anonymous reviewers for their useful suggestions, and Thomas Bartenstein, Michael Carbin, Juan Chen, Xinyu Feng, Kanad Ghose, Jan Hoffmann, Suresh Jagannathan, Frank Lu, Andrew Myers, Sasa Misailovic, Jens Palsberg, Adrian Sampson, Scott Smith, and Jan Vitek for useful discussions. This work is supported by NSF CAREER Award CCF-1054515 and a Google Faculty Award.

References

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

[2] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 213–227, 1989.

[3] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173, 1994.

[4] Woongki Baek and Trishul M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI '10: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, pages 198–209, 2010.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, 2003.

[6] Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *PLDI*, pages 169–180, 2012.

[7] Anantha P. Chandrakasan, Samuel Sheng, and Robert W. Brodersen. Low power cmos digital design. *IEEE Journal of Solid State Circuits*, 27:473–484, 1992.

[8] Ashutosh S. Dhodapkar and James E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *ISCA '02: Proceedings of the 29th annual international symposium on Computer architecture*, pages 233–244, Washington, DC, USA, 2002. IEEE Computer Society.

[9] Carla Schlatter Ellis. The case for higher-level power management. In *HOTOS '99: Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*, page 162, Washington, DC, USA, 1999. IEEE Computer Society.

[10] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture support for disciplined approximate programming. In *ASPLOS'12*, pages 301–312, 2012.

[11] Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-guided static typing for dynamic scripting languages. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 283–300, 2009.

[12] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. Multivariate amortized resource analysis. In *POPL '11*, pages 357–370, 2011.

[13] Chung-Hsing Hsu and Ulrich Kremer. The design, implementation, and evaluation of a compiler algorithm for cpu energy reduction. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 38–48, New York, NY, USA, 2003. ACM.

[14] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight java - a minimal core calculus for java and gj. In *TOPLAS*, pages 132–146, 1999.

[15] Atsushi Igarashi and Mirko Viroli. Variant parametric types: A flexible subtyping scheme for generics. *TOPLAS*, 28(5):795–847, 2006.

[16] Canturk Isci, Gilberto Contreras, and Margaret Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 359–370, Washington, DC, USA, 2006. IEEE Computer Society.

[17] Canturk Isci and Margaret Martonosi. Identifying program power phase behavior using power vectors. In *Workshop on Workload Characterization*, 2003.

[18] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye. Influence of compiler optimizations on system power. In *DAC '00: Proceedings of the 37th Annual Design Automation Conference*, pages 304–307, New York, NY, USA, 2000. ACM.

[19] Stefanos Kaxiras and Margaret Martonosi. *Computer Architecture Techniques for Power-Efficiency*. Morgan and Claypool Publishers, 1st edition, 2008.

[20] Kenneth Knowles and Cormac Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 32(2):1–34, 2010.

[21] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. Flikker: saving dram refresh-power through critical data partitioning. In *ASPLOS*, pages 213–224, 2011.

[22] Yung-Hsiang Lu, Luca Benini, and Giovanni De Micheli. Requester-aware power reduction. In *In Proceedings of the International Symposium on System Synthesis*, pages 18–24, 2000.

[23] Grigorios Magklis, Michael L. Scott, Greg Semeraro, David H. Albonesi, and Steven Dropsho. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *ISCA '03*, pages 14–27, 2003.

[24] Geoffrey Mainland, Greg Morrisett, and Matt Welsh. Flask: staged functional programming for sensor networks. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 335–346, 2008.

[25] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *TOSEM*, 14:1–41, January 2005.

[26] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin Rinard. Quality of service profiling. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 25–34, 2010.

[27] James Noble, John Potter, and Jan Vitek. Flexible alias protection. In *ECOOP'98*, Brussels, Belgium, July 1998.

[28] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *CC'03*, pages 138–152, April 2003.

[29] Trevor Pering, Tom Burd, and Robert Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *ISLPED '98: Proceedings of the 1998 international symposium on Low power electronics and design*, pages 76–81, 1998.

[30] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate Data Types for Safe and General Low-Power Computation. In *Programming Language Design and Implementation (PLDI)*, June 2011.

[31] Timothy Sherwood, Suleyman Sair, and Brad Calder. Phase tracking and prediction. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 336–349, New York, NY, USA, 2003. ACM.

[32] Jeremy Siek and Walid Taha. Gradual typing for objects. In *ECOOP '07: Proceedings of the 21st European conference on ECOOP 2007*, pages 2–27, Berlin, Heidelberg, 2007. Springer-Verlag.

[33] V. Silva. *Pro Android Games*. Apress Series. Apress, 2010.

[34] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. Eon: a language and runtime system for perpetual systems. In *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 161–174, 2007.

[35] Ching-Long Su and Alvin M. Despain. Cache design tradeoffs for power and performance optimization: a case study. In *ISLPED '95: Proceedings of the 1995 international symposium on Low power design*, pages 63–68, 1995.

[36] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 1997.

[37] Energy Types. `http://www.cs.binghamton.edu/~davidL/et/`.

[38] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced cpu energy. In *OSDI '94: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 2, Berkeley, CA, USA, 1994. USENIX Association.

[39] Qiang Wu, Philo Juang, Margaret Martonosi, and Douglas W. Clark. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. In *ASPLOS-XI*, pages 248–259, 2004.

[40] Fen Xie, Margaret Martonosi, and Sharad Malik. Compile-time dynamic voltage scaling settings: opportunities and limits. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 49–62, 2003.

[41] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *in Proc. IEEE Computer Security Foundations Workshop*, pages 15–23. IEEE Computer Society Press, 2001.

A. Additional Definitions

The `mtype`, `mbody`, and `fields` omitted in the main text of the draft are defined in Fig. 14.

$$\text{(D-MTClass)}\ \frac{\textbf{class}\ \mathtt{c}\ \Delta'\ \textbf{extends}\ \tau'\ \{\overline{M}..\} \in \Psi \qquad \Delta' = \iota'; \Sigma \qquad \Delta\ \tau\ \mathtt{md}(\overline{\tau}\ \overline{\mathtt{x}})\{e\}\ \in\ \overline{M}}{\mathtt{mtype}(\mathtt{md}, \mathtt{c}\langle\iota\rangle)\ \stackrel{\text{def}}{=}\ (\Delta.(\overline{\tau} \to \tau))\{\iota/\iota'\}}$$

$$\text{(D-MTSuper)}\ \frac{\textbf{class}\ \mathtt{c}\ \Delta'\ \textbf{extends}\ \tau'\ \{\overline{M}..\} \in \Psi \qquad \Delta' = \iota'; \Sigma \qquad \Delta\ \tau\ \mathtt{md}(\overline{\tau}\ \overline{\mathtt{x}})\{e\}\ \notin\ \overline{M}}{\mathtt{mtype}(\mathtt{md}, \mathtt{c}\langle\iota\rangle)\ \stackrel{\text{def}}{=}\ \mathtt{mtype}(\mathtt{md}, \tau'\{\iota/\iota'\})}$$

$$\text{(D-MBClass)}\ \frac{\textbf{class}\ \mathtt{c}\ \Delta_0\ \textbf{extends}\ \tau'\ \{\overline{M}..\} \in \Psi \qquad \Delta_0 = \iota_0'; \Sigma_0 \qquad \Delta\ \tau\ \mathtt{md}(\overline{\tau}\ \overline{\mathtt{x}})\{e\}\ \in\ \overline{M} \qquad \Delta = \iota'; \Sigma'}{\mathtt{mbody}(\mathtt{md}\langle\iota\rangle, \mathtt{c}\langle\iota_0\rangle)\ \stackrel{\text{def}}{=}\ \overline{\mathtt{x}}.e\{\iota_0/\iota_0'\}\{\iota/\iota'\}}$$

$$\text{(D-MBSuper)}\ \frac{\textbf{class}\ \mathtt{c}\ \Delta\ \textbf{extends}\ \tau'\ \{\overline{M}..\} \in \Psi \qquad \Delta_0 = \iota_0'; \Sigma_0 \qquad \Delta\ \tau\ \mathtt{md}(\overline{\tau}\ \overline{\mathtt{x}})\{e\}\ \notin\ \overline{M}}{\mathtt{mbody}(\mathtt{md}\langle\iota\rangle, \mathtt{c}\langle\iota_0\rangle)\ \stackrel{\text{def}}{=}\ \mathtt{mbody}(\mathtt{md}\langle\iota\rangle, \tau'\{\iota_0/\iota_0'\})}$$

$$\text{(D-FClass)}\ \frac{\textbf{class}\ \mathtt{c}\ \Delta\ \textbf{extends}\ \tau\ \{\overline{F}; ..\}\ \in \Psi \qquad \Delta = \iota'; \Sigma \qquad \mathtt{fields}(\tau\{\iota/\iota'\}) = \overline{F}'}{\mathtt{fields}(\mathtt{c}\langle\iota\rangle) \triangleq \overline{F}' \uplus \overline{F}\{\iota/\iota'\}}$$

$$\text{(D-FBase)}\ \mathtt{fields}(\mathtt{Object}\langle\phi; \mu\rangle) \triangleq \epsilon$$

Figure 14. FGJ-like Definitions