Using Dependent Types to Define Energy Augmented Semantics of Programs

Bernard van Gastel[1,2(✉)], Rody Kersten[3], and Marko van Eekelen[1,2]

[1] Institute for Computing and Information Sciences,
Radboud University, Nijmegen, The Netherlands
{b.vangastel,m.vaneekelen}@cs.ru.nl
[2] Faculty of Management, Science and Technology,
Open University of the Netherlands, Heerlen, The Netherlands
{bernard.vangastel,marko.vaneekelen}@ou.nl
[3] Carnegie Mellon Silicon Valley, Moffett Field, CA, USA
rody.kersten@sv.cmu.edu

Abstract. Energy is becoming a key resource for IT systems. Hence, it can be essential for the success of a system under development to be able to derive and optimise its resource consumption. For large IT systems, compositionality is a key property in order to be applicable in practice. If such a method is hardware parametric, the effect of using different algorithms or running the same software on different hardware configurations can be studied. This article presents a hardware-parametric, compositional and precise type system to derive energy consumption functions. These energy functions describe the energy consumption behaviour of hardware controlled by the software. This type system has the potential to predict energy consumptions of algorithms and hardware configurations, which can be used on design level or for optimisation.

1 Introduction

Green computing is an important emerging field within computer science. Much attention is devoted to developing energy-efficient systems, with a traditional focus on hardware. However, this hardware is controlled by software, which therefore has an energy-footprint as well.

A few options are available to a programmer who wants to write energy-efficient code. First, the programmer can look for programming guidelines and design patterns, which generally produce more energy-efficient programs, e.g. [1]. Then, he/she might make use of a compiler that optimizes for energy-efficiency, e.g. [2]. If the programmer is lucky, there is an energy analysis available for his specific platform, such as [3] for a processor that is modeled in SimpleScalar (note that this only analyses the energy consumption of the processor, no other components). However, for most platforms this is not a viable option. In that case, the programmer might use dynamic analysis with a measurement set-up. This, however, is not a trivial task and requires a complex set-up [4]. Moreover, it only yields information for a specific test case.

M. van Eekelen and U. Dal Lago (Eds.): FOPARA 2015, LNCS 9964, pp. 20–39, 2016.
DOI: 10.1007/978-3-319-46559-3_2

Our Approach. We propose a dependent type system, which can be used to analyse energy consumption of software, with respect to a set of hardware component models. Such models can be constructed once, using a measurement set-up. Alternatively, they might be derived from hardware specifications. This type system is precise, in the sense that no over-approximation is used. By expressing energy analysis as a dependent type system, one can easily reuse *energy signatures* for functions which were derived earlier. This makes this new dependent type system modular and composable. Furthermore, the use of energy signatures that form a precise description of energy consumption can be a flexible, modular basis for various kinds of analyses and estimations using different techniques (e.g. lower or upper bound static analysis using approximations or average case analysis using dynamic profiling information).

The presented work is related to our results in [5], where we present an over-approximating energy analysis for software, parametric with respect to a hardware model. That analysis is based on an energy-aware Hoare logic and operates on a simple while-language. It is implemented in the tool ECAlogic [6]. This previous work poses many limitations on hardware models in order to over-approximate and requires re-analysis of the body of a function *at each function call*.

The most important contributions of this article are:

1. A *dependent type system* that, for the analysed code, captures both energy consumption and the effect on the state of hardware components.
2. Through the use of energy type signatures the system is *compositional*, making it more suitable for larger IT systems.
3. The dependent type system derives *precise information*. This in contrast to the energy aware Hoare logic in [5], which uses over-approximations for conditionals and loops.
4. Compared with [5] many restrictions on component models and component functions are now removed. Effectively all that is required now, is that the power consumption of a device can be modelled by a finite state machine in which states correspond to level of power draw and state transitions correspond to changes of power draw level. State transitions occur due to (external) calls of component functions. The state transition itself may consume a certain amount of energy. This makes the system very suited for control software for various devices where the actuators are performed directly without the need for synchronisation.
5. The dependently typed energy signatures can form a solid, modular basis for various approximations with various different static or dynamic techniques.
6. The artificial environment ρ that was introduced in [5] to incorporate user verified properties, is not needed in this dependent type setting. Using dependent types, such properties can be incorporated in a more *elegant* way.

We start the paper with a discussion of the considered language and its (energy) semantics in Sect. 2. Next, we need a type system to express every variable in terms of input variables in the appropriate scope (block, function or

program). We introduce a basic dependent type system in Sect. 3 specifically for this problem. Continuing in Sect. 4, we introduce the main type system which derives from each statement and expression both an energy bound and a component state effect. To demonstrate the type system we analyse and compare two example programs in Sect. 5. We conclude this article with a discussion, describing future work, and a conclusion.

2 Hybrid Modelling: Language and Semantics

Modern electronic systems typically consist of both hardware and software. As we aim to analyse energy consumption of such *hybrid* systems, we consider hardware and software in a single modelling framework. Software plays a central role, controlling various hardware components. Our analysis works on a simple, special purpose language, called ECA. This language has a special construct for calling functions on hardware components. It is assumed that models exist for all the used hardware components, which model the energy consumption characteristics of these components, as well as the state changes induced by and return values of component function calls.

The ECA language is described in Sect. 2.1. Modelling of hardware components is discussed in Sect. 2.2. Energy-aware semantics for ECA are discussed in Sect. 2.3.

2.1 ECA Language

The grammar for the ECA language is defined below. We presume there is a way to differentiate between identifiers that represents variables (Var), function names (FunName), components (Component), and constants (Const).

⟨FunDef⟩ ::= '`function`' ⟨FunName⟩ '`(`' ⟨Var⟩ '`)`' '`begin`' ⟨Expr⟩ '`end`'

⟨Bin-Op⟩ ::= '`+`' | '`-`' | '`*`' | '`>`' | '`>=`' | '`==`' | '`!=`' | '`<=`' | '`<`' | '`and`' | '`or`'

⟨Expr⟩ ::= ⟨Const⟩ | '`-`' ⟨Const⟩ | ⟨Input⟩ | ⟨Var⟩
| ⟨Var⟩ '`:=`' ⟨Expr⟩ | ⟨Expr⟩ ⟨Bin-Op⟩ ⟨Expr⟩
| ⟨Component⟩ '`.`' ⟨FunName⟩ '`(`' ⟨Expr⟩ '`)`'
| ⟨FunName⟩ '`(`' ⟨Expr⟩ '`)`'
| ⟨Stmt⟩ '`,`' ⟨Expr⟩

⟨Stmt⟩ ::= '`skip`' | ⟨Stmt⟩ '`;`' ⟨Stmt⟩ | ⟨Expr⟩
| '`if`' ⟨Expr⟩ '`then`' ⟨Stmt⟩ '`else`' ⟨Stmt⟩ '`end`'
| '`repeat`' ⟨Expr⟩ '`begin`' ⟨Stmt⟩ '`end`'
| ⟨FunDef⟩ ⟨Stmt⟩

The only supported type in the ECA language is a signed integer. There are no explicit booleans. The value 0 is handled as *false*, any other value is handled as *true*. The absence of global variables and the by-value passing of variables to functions imply that functions do not have side-effects on the program state. Functions are statically scoped. Recursion is not currently supported.

The language has an explicit construct for operations on hardware components (e.g. memory, storage or network devices). This allows us to reason about components in a straight-forward manner. Functions on components have a single parameter and always return a value. The notation $C.f$ refers to a function f of a component C.

The language supports a repeat construct, which makes the bound an obvious part of the loop and removes the need for evaluating the loop guard with every iteration (as with the while construct). The loop bound is evaluated once before executing the loop bodies. The repeat construct is used for presentation purposes, without loss of generality, since the more commonly used while construct has the same expressive power, but is more complex to present.

2.2 Hardware Component Modelling

In order to reason about hybrid systems, we need to model hardware components. A component model consists of a *component state* and a set of *component functions* which can change the component state. A component model must capture the behaviour of the hardware component with respect to energy consumption. Component models are used to derive dependent types signifying the energy consumption of the modelled hybrid system. The model can be based on measurements or detailed hardware specifications. Alternatively, a generic component model might be used (e.g. for a generic hard disk drive).

A component state $C.s$ is a collection of variables of any type. They can signify e.g. that the component is on, off or in stand-by. A component function is modelled by two functions: one that produces the return value (rv_f) and one that updates the (component) state (δ_f). A component can have multiple component functions. Any state change in components can only occur during a component function call and therefore is made explicit in the source code.

Each component has a power draw, which depends on the component state. The function ϕ in the component model maps the component state to a power draw. The result of this function is used to calculate *time-dependent energy consumption* in the td_s function. Function td_s gets the elapsed time and the set of component models as input and results in a function that updates the component states.

2.3 Energy-Aware Semantics

We use a fairly standard semantics for our ECA language, to which we add energy-awareness. We do not list the basic semantics here. The energy-aware semantics are given in Fig. 1. The interested reader might refer to [7] for a listing of the full non-energy-aware semantics of a previous version of ECA.

Components consume energy in two distinct ways. A component function might directly induce a *time-independent* energy consumption. Apart from that, it can change the state of the component, affecting its *time-dependent* energy consumption. To be able to calculate time-dependent consumption, we need a

$$\frac{}{\Delta^s \vdash \langle i,\ \sigma,\ \Gamma\rangle \xrightarrow{e} \langle \mathcal{I}(i),\ \sigma,\ td_s(\Gamma, t_{\text{input}})\rangle}\ (\text{esInput}) \qquad \frac{}{\Delta^s \vdash \langle c,\ \sigma,\ \Gamma\rangle \xrightarrow{e} \langle \mathcal{Z}(c),\ \sigma,\ td_s(\Gamma, t_{\text{const}})\rangle}\ (\text{esConst})$$

$$\frac{}{\Delta^s \vdash \langle x,\ \sigma,\ \Gamma\rangle \xrightarrow{e} \langle \sigma(x),\ \sigma,\ td_s(\Gamma, t_{\text{var}})\rangle}\ (\text{esVar})$$

$$\frac{\Delta^s \vdash \langle e_1,\ \sigma,\ \Gamma\rangle \xrightarrow{e} \langle n,\ \sigma',\ \Gamma'\rangle \qquad \Delta^s \vdash \langle e_2,\ \sigma',\ \Gamma'\rangle \xrightarrow{e} \langle m,\ \sigma'',\ \Gamma''\rangle \qquad n \boxdot m = p}{\Delta^s \vdash \langle e_1 \boxdot e_2,\ \sigma,\ \Gamma\rangle \xrightarrow{e} \langle p,\ \sigma'',\ td_s(\Gamma'', t_{\boxdot})\rangle}\ (\text{esBinOp})$$

$$\frac{\Delta^s \vdash \langle e_1,\ \sigma,\ \Gamma\rangle \xrightarrow{e} \langle n,\ \sigma',\ \Gamma'\rangle}{\Delta^s \vdash \langle x{:=}e_1,\ \sigma,\ \Gamma\rangle \xrightarrow{e} \langle n,\ \sigma'[x \leftarrow n],\ td_s(\Gamma', t_{\text{assign}})\rangle}\ (\text{esAssign})$$

$$\frac{\Delta^s \vdash \langle e_1,\ \sigma,\ \Gamma\rangle \xrightarrow{e} \langle a,\ \sigma',\ \Gamma'\rangle \qquad \Gamma'' = td_s(\Gamma', t)}{\Delta^s, C.f = (t, \delta, rv) \vdash \langle C.f(e_1),\ \sigma,\ \Gamma\rangle \xrightarrow{e} \langle rv(\Gamma''(C), a),\ \sigma',\ \Gamma''[C \leftarrow \delta(\Gamma''(C), a)]\rangle}\ (\text{esCallCmpF})$$

$$\frac{\Delta^s \vdash \langle e_1,\ \sigma,\ \Gamma\rangle \xrightarrow{e} \langle a,\ \sigma',\ \Gamma'\rangle \qquad \Delta^s \vdash \langle e_f,\ [x \mapsto a],\ \Gamma'\rangle \xrightarrow{e} \langle n,\ \sigma'',\ \Gamma''\rangle}{\Delta^s, f = (e_f, x) \vdash \langle f(e_1),\ \sigma,\ \Gamma\rangle \xrightarrow{e} \langle n,\ \sigma',\ \Gamma''\rangle}\ (\text{esCallF})$$

$$\frac{\Delta^s \vdash \langle S_1,\ \sigma,\ \Gamma\rangle \xrightarrow{s} \langle \sigma',\ \Gamma'\rangle \qquad \Delta^s \vdash \langle e_1,\ \sigma',\ \Gamma'\rangle \xrightarrow{e} \langle n,\ \sigma'',\ \Gamma''\rangle}{\Delta^s \vdash \langle S_1, e_1,\ \sigma,\ \Gamma\rangle \xrightarrow{e} \langle n,\ \sigma'',\ \Gamma''\rangle}\ (\text{esExprConcat})$$

$$\frac{\Delta^s \vdash \langle e_1,\ \sigma,\ \Gamma\rangle \xrightarrow{e} \langle n,\ \sigma',\ \Gamma'\rangle}{\Delta^s \vdash \langle e_1,\ \sigma,\ \Gamma\rangle \xrightarrow{s} \langle \sigma',\ \Gamma'\rangle}\ (\text{esExprAsStmt}) \qquad \frac{}{\Delta^s \vdash \langle \mathsf{skip},\ \sigma,\ \Gamma\rangle \xrightarrow{s} \langle \sigma,\ \Gamma\rangle}\ (\text{esSkip})$$

$$\frac{\Delta^s \vdash \langle S_1,\ \sigma,\ \Gamma\rangle \xrightarrow{s} \langle \sigma',\ \Gamma'\rangle \qquad \Delta^s \vdash \langle S_2,\ \sigma',\ \Gamma'\rangle \xrightarrow{s} \langle \sigma'',\ \Gamma''\rangle}{\Delta^s \vdash \langle S_1; S_2,\ \sigma,\ \Gamma\rangle \xrightarrow{s} \langle \sigma'',\ \Gamma''\rangle}\ (\text{esStmtConcat})$$

$$\frac{\Delta^s \vdash \langle e_1,\ \sigma,\ \Gamma\rangle \xrightarrow{e} \langle 0,\ \sigma',\ \Gamma'\rangle \qquad \Delta^s \vdash \langle S_2,\ \sigma',\ td_s(\Gamma', t_{\text{if}})\rangle \xrightarrow{s} \langle \sigma'',\ \Gamma''\rangle}{\Delta^s \vdash \langle \mathsf{if}\ e_1\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{end},\ \sigma,\ \Gamma\rangle \xrightarrow{s} \langle \sigma'',\ \Gamma''\rangle}\ (\text{esIf-False})$$

$$\frac{\Delta^s \vdash \langle e_1,\ \sigma,\ \Gamma\rangle \xrightarrow{e} \langle n,\ \sigma',\ \Gamma'\rangle \qquad n \neq 0 \qquad \Delta^s \vdash \langle S_1,\ \sigma',\ td_s(\Gamma', t_{\text{if}})\rangle \xrightarrow{s} \langle \sigma'',\ \Gamma''\rangle}{\Delta^s \vdash \langle \mathsf{if}\ e_1\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \mathsf{end},\ \sigma,\ \Gamma\rangle \xrightarrow{s} \langle \sigma'',\ \Gamma''\rangle}\ (\text{esIf-True})$$

$$\frac{n \leq 0}{\Delta^s \vdash \langle (S_1, n),\ \sigma,\ \Gamma\rangle \xrightarrow{s} \langle \sigma,\ \Gamma\rangle}\ (\text{esRepeatBase})$$

$$\frac{n > 0 \qquad \Delta^s \vdash \langle S_1,\ \sigma,\ \Gamma\rangle \xrightarrow{s} \langle \sigma',\ \Gamma'\rangle \qquad \Delta^s \vdash \langle (S_1, n-1),\ \sigma',\ \Gamma'\rangle \xrightarrow{s} \langle \sigma'',\ \Gamma''\rangle}{\Delta^s \vdash \langle (S_1, n),\ \sigma,\ \Gamma\rangle \xrightarrow{s} \langle \sigma'',\ \Gamma''\rangle)}\ (\text{esRepeatLoop})$$

$$\frac{\Delta^s \vdash \langle e_1,\ \sigma,\ \Gamma\rangle \xrightarrow{e} \langle n,\ \sigma',\ \Gamma'\rangle \qquad \Delta^s \vdash \langle (S_1, n),\ \sigma',\ \Gamma'\rangle \xrightarrow{s} \langle \sigma'',\ \Gamma''\rangle}{\Delta^s \vdash \langle \mathsf{repeat}\ e_1\ \mathsf{begin}\ S_1\ \mathsf{end},\ \sigma,\ \Gamma\rangle \xrightarrow{s} \langle \sigma'',\ \Gamma''\rangle}\ (\text{esRepeat})$$

$$\frac{\Delta^s, f = (e_1, x) \vdash \langle S_1,\ \sigma,\ \Gamma\rangle \xrightarrow{s} \langle \sigma',\ \Gamma'\rangle}{\Delta^s \vdash \langle \mathsf{function}\ f(x)\ \mathsf{begin}\ e_1\ \mathsf{end}\ S_1,\ \sigma,\ \Gamma\rangle \xrightarrow{s} \langle \sigma',\ \Gamma'\rangle}\ (\text{esFuncDef})$$

Fig. 1. Energy-aware semantics. Note that i stands for an INPUT term, c for a CONST term, x for a VAR term, e for a EXPR term, S for a STMT term, $\boxdot$ for a BIN-OP, f for a FUNNAME, and C for a COMPONENT.

basic timing analysis. Each construct in the language therefore has an associated execution time, for instance t_{if}. Component functions have a constant time consumption that is part of its function signature, along with δ and rv, as described in Sect. 2.2.

The function environment Δ^s (s for semantics) contains both the aforementioned component function signatures (triples of duration t, state transition function δ and return value function rv) and function definitions (pairs of function body e and parameter x). Component models are collected in component state environment Γ. Time-dependent energy consumption is accounted for separately in Γ for each component by the td_s function, which gets Γ and a time period t as input and results in a new component state environment. Time independent energy usage can be accounted for by also assigning a constant energy cost to component state update function δ.

We differentiate two kinds of reductions. Expressions reduce from a triple of the expression, program state σ and component state environment Γ, to a triple of a value, a new program state and a new component state environment, with the $\xrightarrow{e}$ operator. As statements do not result in a value, they reduce from a triple of the statement, σ, and Γ to a pair of a new program state and a new component state environment, with the $\xrightarrow{s}$ operator.

We use the following notation for substitution: $\sigma[x \leftarrow a]$. With $[x \mapsto a]$, we construct a new environment in which x has the value a.

There are three rules for the repeat loop. The one labelled *esRepeat* calculates the value of expression e, i.e. the number of iterations for the loop. The *esRepeatLoop* rule then handles each iteration, until the number of remaining iterations is 0. At that point, the evaluation of the loop is ended with the *esRepeatBase* rule. We use a tuple with as first argument the statement to be evaluated and as second the number of times the statement should be evaluated to differentiate it from normal evaluation rules.

3 Basic Dependent Type System

Before we can introduce a dependent type system that can be used to derive energy consumption expressions of an ECA program, we need to define a standard dependent type system to reason about variable values. We separately introduce these dependent type systems for presentation purposes. We describe the dependent type system deriving types signifying energy consumption in Sect. 4.

Note that, instead of direct expressions over inputs, we derive functions that calculate a value based on values of the input. This allows us to combine these functions using function composition and to reuse them as signatures for methods and parts of the code.

We will start with a series of definitions. A program state environment called $PState$ is a function from a variable identifier to a value, i.e. $PState : Var \mapsto Value$. Values are of type $\mathbb{Z}$, like the data type in ECA. A component state $CState$ is collection of states of components that the component function can work on. A *value function* is a function that, given a (program and component) state, will yield a concrete value, i.e. its signature is $PState \times CState \rightarrow Value$. A *state update function* is a function from $PState \times CState$ to $PState \times CState$. Such a function expresses changes to the state, caused by the execution of statements or expressions.

$$\frac{}{\Delta^v \vdash i : \langle Lookup_i,\ id\rangle}\ (\text{btInput}) \qquad \frac{}{\Delta^v \vdash c : \langle Const_{\mathcal{N}(c)},\ id\rangle}\ (\text{btConst})$$

$$\frac{}{\Delta^v \vdash x : \langle Lookup_x,\ id\rangle}\ (\text{btVar})$$

$$\frac{\Delta^v \vdash e_1 : \langle V_1,\ \Sigma_1\rangle \qquad \Delta^v \vdash e_2 : \langle V_2,\ \Sigma_2\rangle \qquad \boxdot \in \textsc{Bin-Op}}{\Delta^v, \Sigma \vdash e_1 \boxdot e_2 : \langle V_1\ \overline{\boxdot}\ (\Sigma_1 \ggg V_2),\ \Sigma_1 \ggg \Sigma_2\rangle}\ (\text{btBinOp})$$

$$\frac{\Delta^v \vdash e : \langle V,\ \Sigma\rangle}{\Delta^v \vdash x{:=}e : \langle V,\ \Sigma \ggg Assign_x(V)\rangle}\ (\text{btAssign})$$

$$\frac{\Delta^v \vdash e : \langle V_{ex},\ \Sigma_{ex}\rangle}{\Delta^v,\ C.f = (x_{f'}, V_{f'}, \Sigma_{f'}) \vdash C.f(e) : \langle \overline{[x_{f'} \leftarrow V_{ex}, \Sigma_{ex}]} \ggg V_{f'},\ Split(\Sigma_{ex}, \overline{[x_{f'} \leftarrow V_{ex}, \Sigma_{ex}]} \ggg \Sigma_{f'})\rangle}\ (\text{btCallCmpF})$$

$$\frac{\Delta^v \vdash e : \langle V_{ex},\ \Sigma_{ex}\rangle}{\Delta^v,\ f = (x_{f'}, V_{f'}, \Sigma_{f'}) \vdash f(e) : \langle \overline{[x_{f'} \leftarrow V_{ex}, \Sigma_{ex}]} \ggg V_{f'},\ Split(\Sigma_{ex}, \overline{[x_{f'} \leftarrow V_{ex}, \Sigma_{ex}]} \ggg \Sigma_{f'})\rangle}\ (\text{btCallF})$$

$$\frac{\Delta^v \vdash S : \Sigma_{st} \qquad \Delta^v \vdash e : \langle V_{ex},\ \Sigma_{ex}\rangle}{\Delta^v \vdash S,e : \langle \Sigma_{st} \ggg V_{ex},\ \Sigma_{st} \ggg \Sigma_{ex}\rangle}\ (\text{btExprConcat})$$

$$\frac{\Delta^v \vdash e : \langle V,\ \Sigma\rangle}{\Delta^v \vdash e : \Sigma}\ (\text{btExprAsStmt}) \qquad \frac{}{\Delta^v \vdash \mathsf{skip} : id}\ (\text{btSkip})$$

$$\frac{\Delta^v \vdash S_1 : \Sigma_1 \qquad \Delta^v \vdash S_2 : \Sigma_2}{\Delta^v \vdash S_1;S_2 : \Sigma_1 \ggg \Sigma_2}\ (\text{btStmtConcat})$$

$$\frac{\Delta^v \vdash e : \langle V_{ex},\ \Sigma_{ex}\rangle \qquad \Delta^v \vdash S_t : \Sigma_t \qquad \Delta^v \vdash S_f : \Sigma_f}{\Delta^v \vdash \mathsf{if}\ e\ \mathsf{then}\ S_t\ \mathsf{else}\ S_f\ \mathsf{end} : if(V_{ex}, \Sigma_{ex} \ggg \Sigma_t, \Sigma_{ex} \ggg \Sigma_f)}\ (\text{btIf})$$

$$\frac{\Delta^v \vdash e : \langle V_{ex},\ \Sigma_{ex}\rangle \qquad \Delta^v \vdash S : \Sigma_{st}}{\Delta^v \vdash \mathsf{repeat}\ e\ \mathsf{begin}\ S\ \mathsf{end} : repeat^v(V_{ex}, \Sigma_{ex}, \Sigma_{st})}\ (\text{btRepeat})$$

$$\frac{\Delta^v \vdash e : \langle V_{ex},\ \Sigma_{ex}\rangle \qquad \Delta^v, f = (x, V_{ex}, \Sigma_{ex}) \vdash S : \Sigma_{st}}{\Delta^v \vdash \mathsf{function}\ f(x)\ \mathsf{begin}\ e\ \mathsf{end}\ S : \Sigma_{st}}\ (\text{btFuncDef})$$

Fig. 2. Basic dependent type system. For expressions it yields a tuple of a *value function* and a *state update function*. For statements the type system only derives a *state update function*.

To make the typing rules more clear, we explain a number of rules in more detail. The full typing rules can be found in Fig. 2. We will start with the variable access rule:

$$\frac{}{\Delta^v \vdash x : \langle Lookup_x,\ id\rangle}\ (\text{btVar})$$

All rules for evaluation of an expression return a tuple of a function returning the value of the expression when evaluated and a function that modifies the program state and component state. The former one captures the value, i.e. it is a state value function. The latter one captures the effect, i.e. it is a state update function. To access a variable, we return the *Lookup* function (defined below), which is parametrised by the variable that is returned. Variable access does not affect the state, hence for that part the identity function *id* is returned.

The *Lookup* function that the *btVar* rule depends on is defined as follows:

$$\begin{aligned} &Lookup_x : PState \times CState \to Value \\ &Lookup_x(ps, cs) = ps(x) \end{aligned}$$

Likewise we need to define a function for the *btConst* rule, dealing with constants:

$$\begin{aligned} &Const_x : PState \times CState \to Value \\ &Const_x(ps, cs) = x \end{aligned}$$

Before we can continue we need to introduce the $\ggg$ operator. We can compose state update functions with the $\ggg$ operator. Note that the composition is reversed with respect to standard mathematical function composition, in order to maintain the order of program execution. For now, T is $PState \times CState$. The $\ggg$ operator can be interpreted as: first apply the effect of the left operand, then execute the right operand on the resulting state. The $\ggg$ operator is defined as:

$$\begin{aligned} \ggg : (PState \times CState \to PState \times CState) \times (PState \times CState \to T) \\ \to (PState \times CState \to T) \\ (A \ggg B)(ps, cs) = B(ps', cs') \text{where} (ps', cs') = A(ps, cs) \end{aligned}$$

Moving on, we can explain the assignment rule below, which assigns a value expressed by expression e to variable x. As an assignment does not modify the value of the expression, the part capturing the value is propagated from the rule deriving the type of x. More interesting is the effect that captures the state change when evaluating the rule. This consists of first applying the state change Σ^v of evaluating the expression e and thereafter replacing the variable x with the result of e (as done by the *Assign* function, defined below). This effect is reached with the $\ggg$ operator.

$$\frac{\Delta^v \vdash e : \langle V,\ \Sigma^v \rangle}{\Delta^v \vdash x{:=}e : \langle V,\ \Sigma^v \ggg Assign_x(V) \rangle} \text{(btAssign)}$$

The operator for assigning a new value to a program variable in the type environment:

$$\begin{aligned} &Assign_x : (PState \times CState \to Value) \to (PState \times CState \to PState \times CState) \\ &Assign_x(V)(ps, cs) = (ps[x \mapsto V(ps, cs)], cs) \end{aligned}$$

In order to support binary operations we need to define a higher order operator $\overline{\boxdot}$ where $\boxdot \in +, -, \times, \div, \ldots$. It evaluates the two arguments and combines the results using $\boxdot$. For now, T is $Value$.

$$\begin{aligned} \overline{\boxdot} : (PState \times CState \to T) \times (PState \times CState \to T) \\ \to (PState \times CState \to T) \\ (A\ \overline{\boxdot}\ B)(ps, cs) = A(ps, cs) \boxdot B(ps, cs) \end{aligned}$$

The binary operation rule can now be introduced. The pattern used in the rule reoccurs multiple times if two expressions (or statements) need to be combined. First the value function representing the first expression, V_1, is evaluated and is combined with the result representing the second expression, V_2, e.g. using the

$\overline{\boxdot}$ operator. But this second value function needs to be evaluated in the right context (state): evaluating the first expression can have side-effects and therefore change the program and component state, and influence the outcome of a value function (as the expression can include function calls, assignments, component function calls, etc.). To evaluate V_2 in the right context we first must apply the side effects of the first expressions using a state update function Σ_1, expressed as $\Sigma_1 \ggg V_2$. We can express the value function that represents the combination of two expressions as $V_1 \overline{\boxdot} (\Sigma_1 \ggg V_2)$. The binary operation rule is defined as:

$$\frac{\Delta^v \vdash e_1 : \langle V_1, \Sigma_1 \rangle \qquad \Delta^v \vdash e_2 : \langle V_2, \Sigma_2 \rangle \qquad \boxdot \in \textsc{Bin-Op}}{\Delta^v, \Sigma \vdash e_1 \boxdot e_2 : \langle V_1 \overline{\boxdot} (\Sigma_1 \ggg V_2), \Sigma_1 \ggg \Sigma_2 \rangle} \text{(btBinOp)}$$

Without explaining the rule in detail, we introduce the conditional operator. The *if* operator captures the behaviour of a conditional inside the dependent type. The first argument denotes a function expressing the value of the conditional. For now, T stands for a tuple $PState \times CState$.

$$\begin{aligned} \mathit{if} : (PState \times CState \to Value) \times (PState \times CState \to T) \\ \times (PState \times CState \to T) \\ \to (PState \times CState \to T) \end{aligned}$$

$$\mathit{if}(c, then, else)(ps, cs) = \begin{cases} then(ps, cs) & \text{if} c(ps, cs) \neq 0 \\ else(ps, cs) & \text{if} c(ps, cs) = 0 \end{cases}$$

The *btRepeat* rule can be used to illustrate a more complex rule. The effect of repeat is the composition of evaluating the bound and evaluating the body a number of times. The latter is captured in a *repeat* function that is defined below. The resulting effect can be defined as follows:

$$\frac{\Delta^v \vdash e : \langle V_{ex}, \Sigma^v_{ex} \rangle \qquad \Delta^v \vdash S : \Sigma^v_{st}}{\Delta^v \vdash \mathsf{repeat}\ e\ \mathsf{begin}\ S\ \mathsf{end} : repeat^v(V_{ex}, \Sigma^v_{ex}, \Sigma^v_{st})} \text{(btRepeat)}$$

The *repeat* operator needed in the *btRepeat* rule captures the behaviour of a loop inside the dependent type. It gets a function that calculates the number of iterations from a type environment, as well as an environment update function for the loop body, and results in an environment update function that represents the effects of the entire loop. Because the value of the bound must be evaluated in the context before the effect is evaluated, we need an extra function. The actual recursion is in the $repeat^v$ function, also defined below.

$$\begin{aligned} repeat^v : (PState \times CState \to Value) \times (PState \times CState \to PState \times CState) \\ \times (PState \times CState \to PState \times CState) \\ \to (PState \times CState \to PState \times CState) \end{aligned}$$

$$\begin{aligned} repeat^v(c, start, body)(ps, cs) = repeat'^v(c(ps, cs), body, ps', cs') \\ \text{where}(ps', cs') = start(ps, cs) \end{aligned}$$

$$repeat'^{v} : \mathbb{Z} \times (PState \times CState \to PState \times CState) \times PState \times CState \\ \to PState \times CState$$

$$repeat'^{v}(n, body, ps, cs) = \begin{cases} (ps, cs) & \text{if } n \leq 0 \\ repeat'^{v}(n-1, body, ps', cs') & \text{if } n > 0 \\ \quad \text{where} & \\ \quad\quad (ps', cs') = body(ps, cs) & \end{cases}$$

In order to explain the component call rule we need an operator for higher order substitution. This operator creates a new program environment, but retains the component state. It can even update the component state given a Σ function, which is needed because this Σ needs to be evaluated using the original program state. The definition is as follows:

$$\overline{[x \leftarrow V, \Sigma]} : Var \times (PState \times CState \to Value) \\ \times (PState \times CState \to PState \times CState) \\ \to (PState \times CState \to PState \times CState)$$

$$\overline{[x \leftarrow V, \Sigma]}(ps, cs) = ([x \mapsto V(ps, cs)], cs') \text{where}(\_, cs') = \Sigma(ps, cs)$$

We also need an additional operator *Split*, because the program state is isolated but the component state is not. *Split* forks the evaluation into two state update functions and joins the results together. The first argument defines the resulting program state, the second defines the resulting component state.

$$Split : (PState \times CState \to PState \times CState) \\ \times (PState \times CState \to PState \times CState) \\ \to (PState \times CState \to PState \times CState)$$

$$Split(A, B)(ps, cs) = (ps', cs') \quad \text{where}(ps', \_) = A(ps, cs) \\ \text{and}(\_, cs') = B(ps, cs)$$

The component functions and normal functions are stored in the environment Δ^v. For each function (both component and language) a triple is stored, which states the variable name of the argument, a value function that represents the return value, and a state update function that represents the effect on the state of executing the called function. We assume the component function calls are already in the environment, function definitions in the language can be placed in the environment by the function definition rule *btFuncDef*. The component function call can now be easily expressed. Likewise we can define the function call, but for brevity it is omitted in the text.

$$\frac{\Delta^v \vdash e : \langle V_{ex},\ \Sigma_{ex} \rangle}{\Delta^v,\ C.f = (x_{f'}, V_{f'}, \Sigma_{f'}) \vdash \\ C.f(e) : \langle \overline{[x_{f'} \leftarrow V_{ex}, \Sigma_{ex}]} \ggg V_{f'},\ Split(\Sigma_{ex}, \overline{[x_{f'} \leftarrow V_{ex}, \Sigma_{ex}]} \ggg \Sigma_{f'}) \rangle} \text{(btCallCmpF)}$$

$$\frac{}{\Delta^{ec} \vdash i : \langle ..., ..., td^{ec}(t_{\text{input}}) \rangle} \text{ (etInput)} \qquad \frac{}{\Delta^{ec} \vdash c : \langle ..., ..., td^{ec}(t_{\text{const}}) \rangle} \text{ (etConst)}$$

$$\frac{}{\Delta^{ec} \vdash x : \langle ..., ..., td^{ec}(t_{\text{var}}) \rangle} \text{ (etVar)}$$

$$\frac{\Delta^{ec} \vdash e_1 : \langle ..., \Sigma_1, E_1 \rangle \qquad \Delta^{ec} \vdash e_2 : \langle ..., \Sigma_2, E_2 \rangle \qquad \boxdot \in \text{Bin-Op}}{\Delta^{ec} \vdash e_1 \boxdot e_2 : \langle ..., ..., E_1 \mathbin{\overline{+}} (\Sigma_1 \ggg (E_2 \mathbin{\overline{+}} (\Sigma_2 \ggg td^{ec}(t_{\boxdot})))) \rangle} \text{ (etBinOp)}$$

$$\frac{\Delta^{ec} \vdash e : \langle ..., \Sigma, E \rangle}{\Delta^{ec} \vdash x{:=}e : \langle ..., ..., E \mathbin{\overline{+}} (\Sigma \ggg td^{ec}(t_{\text{assign}})) \rangle} \text{ (etAssign)}$$

$$\frac{\Delta^{ec} \vdash e : \langle V_{ex}, \Sigma_{ex}, E_{ex} \rangle}{\begin{array}{l}\Delta^{ec}, C.f = (x_{f'}, \ldots, \ldots, E_{f'}, t_{f'}) \vdash \\ \quad C.f(e) : \langle ..., ..., E_{ex} \mathbin{\overline{+}} (\overline{[x_{f'} \leftarrow V_{ex}, \Sigma_{ex}]} \ggg (td^{ec}(t_{f'}) \mathbin{\overline{+}} E_{f'})) \rangle\end{array}} \text{ (etCallCmpF)}$$

$$\frac{\Delta^{ec} \vdash e : \langle V_{ex}, \Sigma_{ex}, E_{ex} \rangle}{\Delta^{ec}, f = (x_{f'}, \ldots, \ldots, E_{f'}) \vdash f(e) : \langle ..., ..., E_{ex} \mathbin{\overline{+}} (\overline{[x_{f'} \leftarrow V_{ex}, \Sigma_{ex}]} \ggg E_{f'}) \rangle} \text{ (etCallF)}$$

$$\frac{\Delta^{ec} \vdash S : \langle \Sigma_{st}, E_{st} \rangle \qquad \Delta^{ec} \vdash e : \langle ..., ..., E_{ex} \rangle}{\Delta^{ec} \vdash S,e : \langle ..., ..., E_{st} \mathbin{\overline{+}} (\Sigma_{st} \ggg E_{ex}) \rangle} \text{ (etExprConcat)}$$

$$\frac{\Delta^{ec} \vdash e : \langle ..., \Sigma, E \rangle}{\Delta^{ec} \vdash e : \langle \Sigma, E \rangle} \text{ (etExprAsStmt)} \qquad \frac{}{\Delta^{ec} \vdash \mathsf{skip} : \langle \ldots, zero \rangle} \text{ (etSkip)}$$

$$\frac{\Delta^{ec} \vdash S_1 : \langle \Sigma_1, E_1 \rangle \qquad \Delta^{ec} \vdash S_2 : \langle \ldots, E_2 \rangle}{\Delta^{ec} \vdash S_1;S_2 : \langle \ldots, E_1 \mathbin{\overline{+}} (\Sigma_1 \ggg E_2) \rangle} \text{ (etStmtConcat)}$$

$$\frac{\Delta^{ec} \vdash e : \langle V_{ex}, \Sigma_{ex}, E_{ex} \rangle \qquad \Delta^{ec} \vdash S_t : \langle \ldots, E_t \rangle \qquad \Delta^{ec} \vdash S_f : \langle \ldots, E_f \rangle}{\begin{array}{l}\Delta^{ec} \vdash \mathsf{if}\ e\ \mathsf{then}\ S_t\ \mathsf{else}\ S_f\ \mathsf{end} : \\ \quad \langle \ldots, E_{ex} \mathbin{\overline{+}} (\Sigma_{ex} \ggg td^{ec}(t_{\text{if}})) \mathbin{\overline{+}} if(V_{ex}, \Sigma_{ex} \ggg E_t, \Sigma_{ex} \ggg E_f) \rangle\end{array}} \text{ (etIf)}$$

$$\frac{\Delta^{ec} \vdash e : \langle V_{ex}, \Sigma_{ex}, E_{ex} \rangle \qquad \Delta^{ec} \vdash S : \langle \Sigma_{st}, E_{st} \rangle}{\Delta^{ec} \vdash \mathsf{repeat}\ e\ \mathsf{begin}\ S\ \mathsf{end} : \langle \ldots, E_{ex} \mathbin{\overline{+}} repeat^{ec}(V_{ex}, \Sigma_{ex}, E_{st}, \Sigma_{st}) \rangle} \text{ (etRepeat)}$$

$$\frac{\Delta^{ec} \vdash e : \langle V_{ex}, \Sigma_{ex}, E_{ex} \rangle \qquad \Delta^{ec}, f = (x, V_{ex}, \Sigma_{ex}, E_{ex}) \vdash S : \langle \Sigma_{st}, E_{st} \rangle}{\Delta^{ec} \vdash \mathsf{function}\ f(x)\ \mathsf{begin}\ e\ \mathsf{end}\ S : \langle E_{st}, \Sigma_{st} \rangle} \text{ (etFuncDef)}$$

Fig. 3. Energy-aware dependent type system. It adds an element to the resulting tuples of Fig. 2 signifying the energy consumption of executing the statement or expression.

4 Energy-Aware Dependent Type System

In this section, we present the complete dependent type system, which can be used to determine the energy consumption of ECA programs. The type system presented here should be viewed as an extension of the basic dependent type system presented in Sect. 3. Each time the three dot symbol $\cdots$ is used, that part of the rule is unchanged with respect to the previous section. The type system adds an element to the results of the previous section, signifying the energy bound. Expressions yield a triple, statements yield a pair. The added element is a function that, when evaluated on a concrete environment and component state, results in a concrete energy consumption or energy cost.

An important energy judgment is the td^{ec} function. To account for time-dependent energy usage, we need this function which calculates the energy usage

of all the components over a certain time period. The td^{ec} function is a higher order function that takes the time it accounts for as argument. It results in a function that is just like the other function calculating energy bounds: it takes two arguments, a $PState$ and a $CState$. The definition is given below. It depends on the power draw function ϕ that maps a component state to an energy consumption (as explained in Sect. 2.2). The definition is as follows:

$$td^{ec} : Time \to (PState \times CState \to EnergyCost)$$
$$td^{ec}(t)(ps, cs) = \textstyle\sum_{e \in cs} \phi(e) \times t$$

We can use this td^{ec} function to define a rule for variable lookup, as the only energy cost that is induced by variable access is the energy used by all components for the duration of the variable access. This is expressed in the *etVar* rule, in which the dots correspond to Fig. 1 (the first dots are $Lookup_x$, the second id).

$$\frac{}{\Delta^{ec} \vdash x : \langle \ldots, \ldots, td^{ec}(t_{\text{var}}) \rangle} \text{(etVar)}$$

Using the $\overline{+}$ operator (introduced as $\overline{\boxdot}$, with type T equal to $EnergyCost$), we can define the energy costs of a binary operation. Although the rule looks complex, it uses pattern introduced in Sect. 3 for the binary operation. Basically, the energy cost of a binary operation is the cost of evaluating the operands, plus the cost of the binary operation itself. The binary operation itself only has an associated run-time and by this means induces energy consumption of components. This is expressed by the $td_{ec}(t_{\boxdot})$ function. For the binary operation rule we need to apply this pattern twice, in a nested manner, as the time-dependent function must be evaluated in the context after evaluating both arguments. The (three) energy consumptions are added by $\overline{+}$. One can express the binary operation rule as:

$$\frac{\Delta^{ec} \vdash e_1 : \langle \ldots, \Sigma_1, E_1 \rangle \qquad \Delta^{ec} \vdash e_2 : \langle \ldots, \Sigma_2, E_2 \rangle}{\Delta^{ec} \vdash e_1 \boxdot e_2 : \langle \ldots, \ldots, E_1 \overline{+} (\Sigma_1 \ggg (E_2 \overline{+} (\Sigma_2 \ggg td^{ec}(t_{\boxdot})))) \rangle} \text{(etBinOp)}$$

Calculating the energy cost of the repeat loop can be calculated by evaluating an energy cost function for each loop iteration (in the right context). We therefore have to modify the $repeat^v$ rule to result in an energy cost, resulting in a new function definition of $repeat^{ec}$:

$$\begin{aligned} repeat^{ec} : (PState \times CState \to Value) &\times (PState \times CState \to PState \times CState) \\ &\times (PState \times CState \to EnergyCost) \\ &\times (PState \times CState \to PState \times CState) \\ &\to (PState \times CState \to EnergyCost) \end{aligned}$$

$$\begin{aligned} repeat^{ec}(c, start, cost, body)(ps, cs) &= repeat'^{ec}(c(ps, cs), cost, body, ps', cs') \\ &\text{where}(ps', cs') = start(ps, cs) \end{aligned}$$

$$repeat'^{ec} : \mathbb{Z} \times (PState \times CState \to EnergyCost) \times (PState \times CState \to PState \times CState) \times PState \times CState \to EnergyCost$$

$$repeat'^{ec}(n, cost, body, ps, cs) = \begin{cases} (0 & \text{if } n \leq 0 \\ repeat'^{ec}(n-1, cost, body, ps', cs') & \text{if } n > 0 \\ \quad +cost(ps, cs) & \\ \qquad \text{where} & \\ \qquad (ps', cs') = body(ps, cs) & \end{cases}$$

Using this $repeat^{ec}$ function, the definition of the rule for the repeat is analogous to the previous definition.

$$\frac{\Delta^{ec} \vdash e : \langle V_{ex},\ \Sigma_{ex},\ E_{ex} \rangle \qquad \Delta^{ec} \vdash S : \langle \Sigma_{st},\ E_{st} \rangle}{\Delta^{ec} \vdash \mathsf{repeat}\ e\ \mathsf{begin}\ S\ \mathsf{end} : \langle \ldots,\ E_{ex} \mathbin{\overline{+}} repeat^{ec}(V_{ex}, \Sigma_{ex}, E_{st}, \Sigma_{st}) \rangle} \text{ (etRepeat)}$$

Next is the component function call. The energy cost of a component function call consists of the time taken to execute this function and of explicit energy cost attributed to this call. The environment Δ^{ec} is extended for each component function $C.f$ with two elements: an energy judgment $E_{f'}$ and a run-time $t_{f'}$. Time independent energy usage can be encoded into this $E_{f'}$ function. For functions defined in the language the derived energy judgement is inserted into the environment. There is no need for these functions for an explicit run-time as this is part of the derived energy judgement. Using the patterns described above the component function call is expressed as:

$$\frac{\Delta^{ec} \vdash e : \langle V_{ex},\ \Sigma_{ex},\ E_{ex} \rangle}{\begin{array}{l}\Delta^{ec},\ C.f = (x_{f'}, \ldots, \ldots, E_{f'}, t_{f'}) \vdash \\ \quad C.f(e) : \langle \ldots,\ \ldots,\ E_{ex} \mathbin{\overline{+}} (\overline{[x_{f'} \leftarrow V_{ex}, \Sigma_{ex}]} \ggg (td^{ec}(t_{f'}) \mathbin{\overline{+}} E_{f'})) \rangle\end{array}} \text{ (etCallCmpF)}$$

5 Example

In this section, we demonstrate our analysis on two example programs, comparing their energy usage. Each ECA language construct has an associated execution time bound. These execution times are used in calculating the energy consumption that is time-dependent. Another source of energy consumption in our modelling is time-independent energy usage, which can also be associated with any ECA construct. Adding these two sources together yields the energy consumption.

Consider the example programs in Listings 1 and 2. Both programs play #n beeps at #hz Hz (#n and #hz are input variables). The effect of the first statement is starting a component named SoundSystem, which models a sound card (actually, functions have a single parameter; this parameter is omitted here

```
1 SoundSystem.on();
2 repeat #n begin
3     SoundSystem.playBeepAtHz(#hz);
4     System.sleep()
5 end;
6 SoundSystem.off()
```

Listing 1. Example program

```
1 repeat #n begin
2     SoundSystem.on();
3     SoundSystem.playBeepAtHz(#hz);
4     SoundSystem.off();
5     System.sleep()
6 end
```

Listing 2. Alternative program

as it is not used). After enabling component SoundSystem, beeps are played by calling the function SoundSystem.playBeepAtHz(). Eventually the device is switched off. The sound system has two states, off $<$ on.

For this example, we will assume a start state in which the component is in the off state and an input $n \in \mathbb{Z}$. Furthermore, the time-independent energy cost of SoundSystem.playBeepAtHz() function is i and a call to this function has an execution time of t seconds. The System.sleep() has no associated (time-independent) energy usage, and takes s seconds to complete. The other component function calls and the loop construct are assumed to have zero execution time and zero (time-independent) energy consumption. While switched on, the component has a power draw of u J/s (or W).

We will start with an intuitive explanation of the energy consumption of the program in Listing 1, then continue by applying the analysis presented in this paper and comparing these results. Quickly calculating the execution time of the program yields a result of $n \times (t+s)$ seconds. As the component is switched on at the start and switched off at the end of the program, the time-dependent energy consumption is $n \times (t + s) \times u$. The time-independent energy usage is equal to the number of calls to transmit, thus $n \times i$, resulting in an energy consumption of $n \times (i + (t + s) \times u)$ J.

Now, we will show the results from our analysis. Applying the energy-aware dependent typing rules ultimately yields an energy consumption function and a state update function. Both take a typing and a component state environment as input. The first rule to apply is *etStmtConcat*, with the SoundSystem.on() call as S_1 and the remainder of the program as S_2. The effect of the component function call, calculated with *etCallCmpF*, is that the component state is increased (as this signifies a higher time-dependent energy usage). This effect is represented in the component state update function $\delta_{SoundSystem.on}$. Since we have assumed that the call costs zero time and has no time-independent energy cost, the resulting energy consumption function is the identity function *id*.

We can now analyse the loop with the *etRepeat* rule. We first derive the type of expression e, which determines the number of iterations of the loop. As the expression is a simple variable access, which we assumed not to have any associated costs, the program state and the component states are not touched. For the result of the expression the type system derives $Lookup_{\#\mathsf{n}}$ as V_{ex} in the repeat rule, which is (a look-up of) n. This is a function that, when given an input environment, calculates a number in $\mathbb{Z}$ which signifies the number of iterations.

Moving on, we analyse the body of the loop. This means we again apply the $etCallCmpF$ rule to determine the resource consumption of the call to SoundSystem.playBeepAtHz(). The call takes time t. Its energy consumption is u if the component is switched on. For ease of presentation, we here represent the component state $\{\mathsf{off}, \mathsf{on}\}$ as a variable $e \in \{0, 1\}$. The time-independent energy usage of the loop body is i. The energy consumption function E_{st} of the body is $i + e \times (t + s) \times u$ J.

We can now combine the results of the analysis of the number of iterations and the resource consumption of the loop body (E_{st}) to calculate the consumption of the entire loop. Basically, the resource consumption of the loop E_{st} is multiplied by the number of iterations V_{ex}. This is done in the function $repeat^{ec}(V_{ex}, \Sigma_{ex}, E_{st}, \Sigma_{st})$, in this case $repeat^{ec}(Lookup_{\#\mathsf{n}}, id, E_s t, id)$. Evaluation after the state update function (corresponding to the SoundSystem.on() call), which will update the state of the sound system to on. Evaluating the sequence of both expressions on the start state results in $n \times (i + (t + s) \times u)$ J, signifying the energy consumption of the code fragment.

Analysing the code fragment listed in Listing 2 will result in a type equivalent to $n \times (i + t \times u)$ J, as the cost of switching the sound system on or off is zero. Given the energy characteristics of the sound system, the second code fragment will have a lower energy consumption. Depending on the realistic characteristics (switching a device on or off normally takes time and energy), a realistic trade-off can be made.

6 Related Work

Much of the research that has been devoted to producing green software is focussed on a very abstract level, defining programming and design patterns for writing energy-efficient code [1,8–11]. In [12], a modular design for energy-aware software is presented that is based on a series of rules on UML schemes. In [13,14], a program is divided into "phases" describing similar behaviour. Based on the behaviour of the software, design level optimisations are proposed to achieve lower energy consumption.

Contrary to the abstract levels of the aforementioned papers, there is also research on producing green software on a very low, hardware-specific level [3,15]. Such analysis methods work on specific architectures ([3] on SimpleScalar, [15] on XMOS ISA-level models), while our approach is hardware-parametric.

Furthermore, there is research on building compilers that optimize code for energy-efficiency. In [2], the implementation of several energy optimization methods, including dynamic voltage scaling and bit-switching minimization, into the GCC compiler is described and evaluated. Iterative compilation is applied to energy consumption in [16]. In [17], a technique is proposed in which register labels are encoded alternatively to minimize switching costs. This saves an average of 4.25 % of energy, without affecting performance. The human-written assembly code for an MP3 decoder is optimized by hand in [18], guided by an energy profiling tool based on a proprietary ARM simulator. In [19], functional

units are disabled to reduce (leakage) energy consumption of a VLIW processor. The same is done for an adaptation of a DEC Alpha 21264 in [20]. Reduced bit-width instruction set architectures are exploited to save energy in [21]. Energy is saved on memory optimizations in [22–25], while [26,27] focus on variable-voltage processors.

A framework called GREEN is presented in [28]. This framework allows programmers to approximate expensive functions and calculate Quality of Service (QoS) statistics. It can thus help leverage a trade-off between performance and energy consumption on the one hand, and QoS on the other.

In [29], Resource-Utilization Models (RUMs) are presented, which are an abstraction of the resource behaviour of hardware components. The component models in this paper can be viewed as an instance of a RUM. RUMs can be analysed, e.g., with the model checker UPPAAL, while we use a dedicated dependent type system. A possible future research direction is to incorporate RUMs into our analysis as component models.

Analyses for consumption of generic resources are built using recurrence relation solving [30], amortized analysis [31], amortization and separation logic [32] and a Vienna Development Method style program logic [33]. The main differences with our work are that we include an explicit hardware model and a context in the form of component states. This context enables the inclusion of power-draw that depends on the state of components..

Several dependently typed programming languages exist, such as EPIGRAM [34] and AGDA [35]. The DEPUTY system, which adds dependent typing to the C language, is described in [36]. Dependent types are applied to security in [37]. Security types there enforce access control and information flow policies.

7 Discussion

The foreseen application area of the proposed analysis is predicting the energy consumption of control systems, in which software controls a number of peripherals. This includes control systems in factories, cars, airplanes, smart-home applications, etc. Examples of hardware components range from a disk drive or sound card, to heaters, engines, motors and urban lighting. The proposed analysis can predict the energy consumption of multiple algorithms and/or different hardware configurations. The choice of algorithm or configuration may depend on the expected workload. This makes the proposed technique useful for both programmers and operators.

The used hardware models can be abstracted, making clear the relative differences between component-methods and component-states. This makes the proposed approach still applicable even when no final hardware component is available for basing the hardware model on, or this model is not yet created. We observe that many decisions are based on relative properties between systems.

The type system derives precise types, in a syntax-directed manner. Soundness and completeness can be proven by induction on the syntax structure of the program. This proof is similar to the proof in [7], but more straightforward due to the absence of approximations.

There are certain properties hardware models must satisfy. Foremost, the models are discrete. Implicit state changes by hardware components cannot be expressed. Energy consumption that gradually increases or decreases over time can therefore not be modelled directly. However, discrete approximations may be used. Compared to the Hoare logic in [5], many restrictions are not present in the proposed type system. Foremost, this type system does not have the limitation that state change cannot depend on the argument of a component function nor that the return value of a component function cannot depend on the state of the component. More realistic models can therefore be used.

The quality of the derived energy expressions is directly related to the quality of the used hardware models. We envision that, in many cases, relative consumption information is sufficient to support design decisions. Depending on the goal, it is possible to use multiple models for one and the same hardware component. For instance, if the hardware model is constructed as a worst-case model, the type system will produce worst-case information. Similarly one can use average-case models to derive average case information.

8 Future Work

In future work, we aim to implement this analysis in order to evaluate its suitability for larger systems and validate practical applicability. We intend to experiment with implementations of various derived approximating analyses in order evaluate which techniques/approximations work best in which context.

A limitation is currently that only a system controlled by *one* central processor can be analysed. Modern systems often consists of a network of interacting systems. Therefore, incorporating interacting systems would increase applicability of the approach.

Another future research direction is to expand the supported input language. Currently, recursion is not supported. An approach to add recursion to the input language is to use the function signatures to compose a set of cost relations (a special case of recurrence relations). A recurrence solver can then eliminate the recursion in the resulting function signatures. In order to support data types, a size analysis of data types is needed to enable iteration over data structures, e.g. using techniques similar to [38,39].

On the language level the type system is precise, however it does not take into account optimisations and transformations below the language level. This can be achieved by analysing the software on a lower level, for example the intermediate representation of a modern compiler. Another motivation to use such an intermediate representation as the input language is support for (combinations of) many higher level languages. In this way, programs written in and consisting of multiple languages can be analysed. It can also account for optimisations (such as common subexpression elimination, inlining, statically evaluating expressions), which in general reduce the execution time of the program and therefore impact the time-dependent energy usage (calls with side effects like component function calls are generally not optimised).

9 Conclusion

The presented type system captures energy bounds for software that is executed on hardware components of which component models are available. It is *precise, modular and elegant*, yet retains the hardware-parametric aspect.

The new type system is in itself *precise*, as there is no over-estimation, as opposed to the Hoare Logic [5], in which the conditional and loop are over-estimated. Also the class of programs that can be studied is larger, as many of the restrictions needed for the over-approximation are now lifted.

The presented hardware-parametric dependent type system with function signatures enables *modularity*. While analysing the energy consumption of an electronic system, instead of re-analysing the body of functions each time a function call is encountered, the function signature is reused.

By using a dependent type system to express all variables in terms of input variables, the resulting approach is *elegant and concise*, as no externally verified properties are needed.

References

1. Saxe, E.: Power-efficient software. Commun. ACM **53**(2), 44–48 (2010)
2. Zhurikhin, D., Belevantsev, A., Avetisyan, A., Batuzov, K., Lee, S.: Evaluating power aware optimizations within GCC compiler. In: GROW-2009: International Workshop on GCC Research Opportunities (2009)
3. Jayaseelan, R., Mitra, T., Li, X.: Estimating the worst-case energy consumption of embedded software. In: Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 81–90. IEEE (2006)
4. Ferreira, M.A., Hoekstra, E., Merkus, B., Visser, B., Visser, J.: SEFLab: A lab for measuring software energy footprints. In: 2013 2nd International Workshop on Green and Sustainable Software (GREENS), pp. 30–37. IEEE (2013)
5. Kersten, R., Toldin, P.P., Gastel, B., Eekelen, M.: A hoare logic for energy consumption analysis. In: Dal Lago, U., Peña, R. (eds.) FOPARA 2013. LNCS, vol. 8552, pp. 93–109. Springer, Heidelberg (2014). doi:10.1007/978-3-319-12466-7_6
6. Schoolderman, M., Neutelings, J., Kersten, R., van Eekelen, M.: ECAlogic: hardware-parametric energy-consumption analysis of algorithms. In: Proceedings of the 13th Workshop on Foundations of Aspect-oriented Languages, FOAL 2014, pp. 19–22. ACM, New York (2014)
7. Parisen Toldin, P., Kersten, R., van Gastel, B., van Eekelen, M.: Soundness proof for a hoare logic for energy consumption analysis. Technical Report ICIS-R13009, Radboud University Nijmegen, October 2013
8. Albers, S.: Energy-efficient algorithms. Commun. ACM **53**(5), 86–96 (2010)
9. Ranganathan, P.: Recipe for efficiency: principles of power-aware computing. Commun. ACM **53**(4), 60–67 (2010)
10. Naik, K., Wei, D.S.L.: Software implementation strategies for power-conscious systems. Mob. Netw. Appl. **6**(3), 291–305 (2001)
11. Sivasubramaniam, A., Kandemir, M., Vijaykrishnan, N., Irwin, M.J.: Designing energy-efficient software. In: International Parallel and Distributed Processing Symposium, Los Alamitos, CA, USA. IEEE Computer Society (2002)

12. te Brinke, S., Malakuti, S., Bockisch, C., Bergmans, L., Akşit, M.: A design method for modular energy-aware software. In: Proceedings of the 28th Annual ACM Symposium on Applied Computing, pp. 1180–1182. ACM, New York (2013)
13. Cohen, M., Zhu, H.S., Senem, E.E., Liu, Y.D.: Energy types. SIGPLAN Not. **47**(10), 831–850 (2012)
14. Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., Grossman, D.: EnerJ: approximate data types for safe and general low-power computation. SIGPLAN Not. **46**(6), 164–174 (2011)
15. Kerrison, S., Liqat, U., Georgiou, K., Mena, A.S., Grech, N., Lopez-Garcia, P., Eder, K., Hermenegildo, M.V.: Energy consumption analysis of programs based on XMOS ISA-level models. In: Gupta, G., Peña, R. (eds.) LOPSTR 2013. LNCS, vol. 8901, pp. 72–90. Springer, Heidelberg (2014)
16. Gheorghita, S.V., Corporaal, H., Basten, T.: Iterative compilation for energy reduction. J. Embedded Comput. **1**(4), 509–520 (2005)
17. Mehta, H., Owens, R.M., Irwin, M.J., Chen, R., Ghosh, D.: Techniques for low energy software. In: ISLPED 1997: Proceedings of the 1997 International Symposium on Low Power Electronics and Design, pp. 72–75. ACM, New York (1997)
18. Šimunić, T., Benini, L., De Micheli, G., Hans, M.: Source code optimization and profiling of energy consumption in embedded systems. In: ISSS 2000: Proceedings of the 13th International Symposium on System Synthesis, Washington, DC, pp. 193–198. IEEE Computer Society (2000)
19. Zhang, W., Kandemir, M., Vijaykrishnan, N., Irwin, M.J., De, V.: Compiler support for reducing leakage energy consumption. In: DATE 2003: Proceedings of the Conference on Design, Automation and Test in Europe, Washington, DC. IEEE Computer Society (2003)
20. You, Y.P., Lee, C., Lee, J.K.: Compilers for leakage power reduction. ACM Trans. Des. Autom. Electron. Syst. **11**(1), 147–164 (2006)
21. Shrivastava, A., Dutt, N.: Energy efficient code generation exploiting reduced bitwidth instruction set architectures (rISA). In: ASP-DAC 2004: Proceedings of the 2004 Asia and South Pacific Design Automation Conference, Piscataway, NJ, USA, pp. 475–477. IEEE Press (2004)
22. Joo, Y., Choi, Y., Shim, H., Lee, H.G., Kim, K., Chang, N.: Energy exploration and reduction of SDRAM memory systems. In: DAC 2002: Proceedings of the 39th Annual Design Automation Conference, pp. 892–897. ACM, New York (2002)
23. Verma, M., Wehmeyer, L., Pyka, R., Marwedel, P., Benini, L.: Compilation and simulation tool chain for memory aware energy optimizations. In: SAMOS, pp. 279–288 (2006)
24. Jones, T.M., O'Boyle, M.F.P., Abella, J., González, A., Ergin, O.: Energy-efficient register caching with compiler assistance. ACM Trans. Archit. Code Optim. **6**(4), 1–23 (2009)
25. Lee, H.G., Chang, N.: Energy-aware memory allocation in heterogeneous non-volatile memory systems. In: ISLPED 2003: Proceedings of the 2003 International Symposium on Low Power Electronics and Design, pp. 420–423. ACM, New York (2003)
26. Okuma, T., Yasuura, H., Ishihara, T.: Software energy reduction techniques for variable-voltage processors. IEEE Des. Test **18**(2), 31–41 (2001)
27. Saputra, H., Kandemir, M., Vijaykrishnan, N., Irwin, M.J., Hu, J.S., Hsu, C.H., Kremer, U.: Energy-conscious compilation based on voltage scaling. LCTES/SCOPES 2002: Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems, pp. 2–11. ACM, New York (2002)

28. Baek, W., Chilimbi, T.M.: Green: a framework for supporting energy-conscious programming using controlled approximation. SIGPLAN Not. **45**(6), 198–209 (2010)
29. te Brinke, S., Malakuti, S., Bockisch, C.M., Bergmans, L.M.J., Akşit, M., Katz, S.: A tool-supported approach for modular design of energy-aware software. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing, Gyeongju, Korea, SAC 2014. ACM, March 2014
30. Albert, E., Arenas, P., Genaim, S., Puebla, G.: Closed-form upper bounds in static cost analysis. J. Autom. Reason. **46**(2), 161–203 (2011)
31. Hoffmann, J., Aehlig, K., Hofmann, M.: Multivariate amortized resource analysis. In: Ball, T., Sagiv, M. (eds.) POPL 2011, pp. 357–370. ACM (2011)
32. Atkey, R.: Amortised resource analysis with separation logic. In: Gordon, A.D. (ed.) ESOP 2010. LNCS, vol. 6012, pp. 85–103. Springer, Heidelberg (2010). doi:10.1007/978-3-642-11957-6_6
33. Aspinall, D., Beringer, L., Hofmann, M., Loidl, H.W., Momigliano, A.: A program logic for resources. Theor. Comput. Sci. **389**(3), 411–445 (2007)
34. McBride, C.: Epigram: practical programming with dependent types. In: Vene, V., Uustalu, T. (eds.) AFP 2004. LNCS, vol. 3622, pp. 130–170. Springer, Heidelberg (2005). doi:10.1007/11546382_3
35. Bove, A., Dybjer, P., Norell, U.: A brief overview of agda - a functional language with dependent types. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 73–78. Springer, Heidelberg (2009)
36. Condit, J., Harren, M., Anderson, Z., Gay, D., Necula, G.C.: Dependent types for low-level programming. In: Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 520–535. Springer, Heidelberg (2007). doi:10.1007/978-3-540-71316-6_35
37. Morgenstern, J., Licata, D.R.: Security-typed programming within dependently typed programming. Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming, ICFP 2010, pp. 169–180. ACM, New York (2010)
38. Shkaravska, O., Eekelen, M., Tamalet, A.: Collected size semantics for strict functional programs over general polymorphic lists. In: Dal Lago, U., Peña, R. (eds.) FOPARA 2013. LNCS, vol. 8552, pp. 143–159. Springer, Heidelberg (2014). doi:10.1007/978-3-319-12466-7_9
39. Tamalet, A., Shkaravska, O., van Eekelen, M.: Size analysis of algebraic data types. In: Achten, P., Koopman, P., Morazán, M. (eds.) Trends in Functional Programming, vol. 9 of Trends in Functional Programming, pp. 33–48. Intellect (2009)

http://www.springer.com/978-3-319-46558-6

Foundational and Practical Aspects of Resource Analysis
4th International Workshop, FOPARA 2015, London, UK, April 11, 2015. Revised Selected Papers
van Eekelen, M.; Dal Lago, U. (Eds.)
2016, IX, 127 p. 30 illus., Softcover
ISBN: 978-3-319-46558-6