Cost Recurrences for DML Programs

Bernd Grobauer

BRICS Report Series RS-01-25

ISSN 0909-0878 June 2001

Cost Recurrences for DML Programs

Bernd Grobauer [*]

BRICS[†]
Department of Computer Science
University of Aarhus

June, 2001

Abstract

A cost recurrence describes an upper bound for the running time of a program in terms of the size of its input. Finding cost recurrences is a frequent intermediate step in complexity analysis, and this step requires an abstraction from data to data size. In this article, we use information contained in dependent types to achieve such an abstraction: Dependent ML (DML), a conservative extension of ML, provides dependent types that can be used to associate data with size information, thus describing a possible abstraction. We systematically extract cost recurrences from first-order DML programs, guiding the abstraction from data to data size with information contained in DML type derivations.

1 Introduction

Analyzing the time complexity of a program is usually carried out in two steps. First, one establishes an upper bound of the program's running time as a function of the size of its input. Second, one approximates the growth of this extracted bounding function, thus determining the complexity class of the program. The first step requires an abstraction from data to data size. Information contained in *dependent types* can be used to achieve such an abstraction. In this article, we show how to automatically extract time bounds from first-order programs written in Dependent ML (DML), an extension of ML that provides a limited form of dependent types. If a bound is successfully extracted, we can guarantee that it is a *recurrence*, i.e., an equation defining a function in terms of its result on smaller inputs. A recurrence that describes an upper bound for the running time of a program is called a *cost recurrence*.

[*]Ny Munkegade, Building 540, 8000 Aarhus C, Denmark.
E-mail: `grobauer@brics.dk`
[†]Basic Research in Computer Science (`www.brics.dk`),
funded by the Danish National Research Foundation.

Limits and achievements of automated cost analysis *Automated* cost analyses have inherent limits. For example, finding a cost recurrence for a program is at least as hard as proving termination of the program. Also, finding good approximations for the growth of recurrences (or, in general, almost any kind of function) is known to be a hard problem.

Nevertheless, automated methods for cost analysis have been proposed. One choice is to restrict the class of programs such that termination is guaranteed, and the extracted time bounds can easily be approximated. For example, Reistad and Gifford [7] consider functional programs without general recursion, using only combinators such as `map` and `zip`. Methods that treat more general programs, as for example proposed by Le Métayer [4] and Rosendahl [8], usually focus on extracting a cost *program* $\mathtt{p}^c$; if $\mathtt{p}^c$ terminates, it calculates an upper bound for the running time of a program `p`. Transforming $\mathtt{p}^c$ may yield a version compact enough to read off the complexity class of `p`. If not, $\mathtt{p}^c$ still may be useful for more empirical attempts to determine the time complexity of `p`, such as plotting input size against the running time calculated by $\mathtt{p}^c$.

Dependent ML DML, which was developed by Xi [12, 16] in his PhD thesis, extends ML with a limited form of dependent types: A DML type can be *enriched* with indices taken from a constraint domain (e.g., integers equipped with their usual operations, with linear (in)equalities as constraints). For example, the data type of lists can be enriched with a notion of length or a data type of trees with a notion of height. The type language is expressive enough to encode well-formedness criteria, such as a tree being balanced. DML function types can express non-trivial properties, for example that a list is always mapped to a list of the same length, or that a function with a balanced tree as input always returns a balanced tree.

The design philosophy of DML is to use type-checking for the verification of non-trivial correctness properties of ML programs—every valid ML program is a valid DML program, because DML extends ML conservatively. For example, to verify that a program for inserting an element into a balanced tree indeed returns a balanced tree, the user needs to (1) enrich a data type of trees such that only balanced trees are accepted, and (2) declare in a type annotation that the insert function maps balanced trees to balanced trees. A range of similar examples convincingly demonstrates that DML is a useful tool for practical programming [12, 13, 15, 16].

This work We use information contained in DML type derivations to extract cost recurrences from DML programs. With DML types, data can be associated with a measure of data size, which essentially describes an abstraction from data to data size that is necessary for extracting a cost recurrence. For example, enriching the data type of lists with a notion of length describes an abstraction from lists to their length. More intricate measures—the high expressiveness of DML types allows the user to tailor measures to each situation. In many cases, measures with several components (e.g., for trees, the pair of height and number of leaves) prove to be useful. Size measures often coincide with shape information for data that is useful for verifying program properties by DML type-checking. Therefore, in many cases, DML types that express correctness properties of a program can be reused for establishing the complexity of the program.

We allow recurrences to contain logical formulas, which are used to restrict arguments that cannot be completely determined. Thus, logical information contained in DML type derivations about such arguments can be included in cost recurrence rather than approximating them in an ad-hoc way. Compared with other methods that extract executable cost bounds—and therefore are required to carry out approximations—we leave the choice of how to approximate to the user, thus separating concerns between extracting a cost recurrence and solving it.

We combine the extraction of a cost bound with a check whether the result is indeed a recurrence: The information contained in DML type derivations facilitates a check of whether the size measure decreases for each recursive call under a wellfounded ordering. In other words, the user has to choose a size measure that constitutes a termination order for the program in question. This is no limitation compared to other methods: Because finding a cost bound entails a termination proof, in all methods for cost analysis a termination proof needs to be found in some way. It is an asset of using DML that the termination proof can be concisely encoded through the size measure.

An example Consider a `zip` function written in ML:

```
fun zip lp =
 case lp of
    (nil,nil) => nil
  | (cons(x,xs),cons(y,ys)) => cons((x,y),zip(xs,ys))
```

DML offers the possibility to annotate `zip` with a type containing an enriched version of lists. We enrich the data type of lists with a notion of length; the type of α-typed lists consisting of n elements is written as $\alpha\,\mathsf{list}(n)$. Obviously, `zip` should take two lists of equal length and return a list of the same length. DML type checking validates that `zip` has the type

$$\Pi\, n : \mathbb{N} \,.\, \alpha\,\mathsf{list}(n) \times \beta\,\mathsf{list}(n) \to (\alpha \times \beta)\mathsf{list}(n).$$

Intuitively, Π can be read as "for all".[1] In ML, a pair of two lists with different lengths could be passed to `zip`, which would result in a runtime error. In DML, the given type of `zip` allows `zip` only to be called with two lists of equal length. The type also shows that the resulting list is of the same length as the input lists.

[1]Formally, Π introduces a dependent product, i.e., a product where the value of the first component (here n) determines the type of the second component (here the function from a pair of lists of length n to a list of length n). Dependent products are also called Π-types.

Let us measure running time as the number of calls to user-defined functions, giving each call a cost of one unit. The resulting recurrence describes the number of calls to `zip` as a function of the length of the two input lists:

$$zip^c(n) = \begin{cases} n = 0 \to 0 \\ n > 0 \to 1 + zip^c(n-1) \end{cases}$$

This cost recurrence is extracted from a DML type derivation for `zip`. In the type derivation, the arguments to `zip`—two lists—are associated with an index n that represents their length. Using this information, the extraction algorithm abstracts from the lists to their length n. For example, the case expression is turned into a conditional by inferring for each branch a condition on n that has to hold if the pattern is matched. Similarly, the algorithm derives from the type derivation that the recursive call of `zip` has a list of length $n - 1$ as argument, and thus generates a call $zip^c(n-1)$. Obviously, $n - 1 < n$, so the extracted bound is a recurrence.

The recurrence can easily be solved: $zip^c(n) = n$.

The remainder of the article The article is structured as follows: Section 2 gives an introduction to DML, Section 3 presents an intuitive account of our method for extracting cost recurrences and gives several examples, Section 4 contains a formal account, Section 5 treats related work, and Section 6 concludes. Appendix A gives a short overview over the formal definition of DML, and Appendix B contains details of the formal development of this work.

2 Background: Dependent ML

DML provides dependent types in which type index objects are limited to some constraint domain C. Type checking for DML is decidable; it is based on solving constraints in C. For dependently typed languages with significantly more expressive types (e.g., Cayenne [1]) type checking is undecidable.

We consider an effect-free fragment of DML. As constraint domain, we choose integers, constrained by linear (in)equalities—we write $\mathbb{Z}$ both for the sort of integers and the constraint domain. In the following, we present a short introduction to programming in DML and sketch the formal specification of DML. The latter forms the basis for the formal development presented in Section 4.

2.1 A programmer's view of DML

The only new aspect for an ML programmer is the extended type system, which contains type indices, in the present case integers.

2.1.1 Enriched recursive data types

As indicated in the example in Section 1, in DML a list type can be enriched with a notion of length, enabling us to express the type of α-typed lists of n

elements as $\alpha\,\mathsf{list}(n)$. The DML data-type definition is

```
datatype α list with ℕ =
   nil(0)
 | Π n : ℕ . cons(n + 1) of α ×  α list(n)
```

It is obtained from an ordinary definition of lists in ML by making the following additions:

1. The phrase "`with` $\mathbb{N}$" has been added. This signifies that the data type of lists is to be enriched with one index and that this index is restricted to the sort of natural numbers. The constraint language of DML allows the definition of subsorts of an already defined sort: $\mathbb{N}$ stands for $\{k : \mathbb{Z} \mid k \geq 0\}$.

2. Constructors and occurrences of "`list`" are augmented with an index. The constructor `nil` is indexed with 0, thus defining the empty list to be of type $\alpha\,\mathsf{list}(0)$. A list built with `cons` is of type $\alpha\,\mathsf{list}(n+1)$, provided that `cons` was applied to an element of type $\alpha\,\mathsf{list}(n)$. Hence, `cons` is indexed with $n+1$.

3. The definition of the `cons` case exhibits a quantification over an index variable n. This index variable is necessary to express the dependence between the index of `cons` and the index of the list appearing in its branch. The quantification restricts the index variable to the sort of natural numbers.

Similarly, we can define the data type of a list of lists $\alpha\,\mathsf{llist}(m,n)$ as

```
datatype α llist with (ℕ,ℕ) =
   lnil(0,0)
 | Π m, n₁, n₂ : ℕ . lcons(m + 1, n₁ + n₂) of
                     α list(n₁) × α llist(m, n₂)
```

The first index stands for the number of inner lists and the second index for the total number of elements the inner lists contain.

The example of lists provides some intuition of how to define enriched recursive data types in two steps: First, decide on the number of indices to be used in the data type, along with the sorts the indices are to be restricted to. Second, annotate each constructor with the appropriate indices. When an index of a constructor depends on indices of recursive data types that appear under that constructor, introduce new index variables using quantification. An index can be defined as a function of other indices using all operations of the constraint domain.

In the introduction we mentioned that enriched data types can encode well-formedness criteria. As an example, we define a data type of height-balanced trees, i.e., the height difference between the two children of a node can be at most one:

```
datatype α HBtree with (ℕ,ℕ) =
   Leaf(0,0)
 | Π s₁, s₂, h₁ : ℕ . Π h₂ : {k : ℕ | h₁ − 1 ≤ k ≤ h₁ + 1} .
    Node(1 + max(h₁, h₂), s₁ + s₂ + 1) of
     α HBtree(h₁,s₁) × α × α HBtree(h₂,s₂)
```

We use two indices, where the first represents the height of the tree and the second represents the number of elements stored in the tree. When defining a node, we require for the heights h_1 and h_2 of the subtrees, that they differ by at most one. This can be achieved by (1) defining a sort of natural numbers k that differ by at most one from h_1, and (2) restricting h_2 to this new sort. As a consequence, only two trees with a height difference of at most one can be the children of a node, i.e., only height-balanced trees can have type $\alpha\,\mathsf{HBtree}$.

Note that for defining a new sort, all predicates and operations of the chosen constraint domain can be used. In the case of height-balanced trees, we use $-$, $+$, and $\leq$.

2.1.2 DML function types

As in ML, a data-type definition gives rise to type declarations for its constructors. For example, the definition of enriched lists presented above yields a type $\alpha\,\mathsf{list}(0)$ for the constructor `nil`, and the type

$$\Pi\, n : \mathbb{N}\,.\, \alpha \times \alpha\,\mathsf{list}(n) \rightarrow \alpha\ \mathsf{list}(n+1)$$

for the constructor `cons`.

Figure 1 shows three functions operating on the data types defined in Section 2.1.1, together with their DML types:

- `append` takes two lists of n_1 and n_2 elements, respectively, and returns a list of $n_1 + n_2$ elements.
- `flatten` takes a list of lists that contain a total number of n elements, and returns a list of n elements
- `occurs` takes a string and a balanced tree, and returns a truth value, according to whether the string is stored in the tree or not (assuming a sorted balanced tree).

The DML types of `append`, `flatten` and `occurs` add shape information to the respective ML type of each function: In the case of `append` and `flatten`, we learn about the shape of the result, i.e., how long the output list is. For `occurs`, the DML type restricts the input tree to trees of a special shape, namely balanced trees.

So far, the output indices in the DML type of a function could always be specified as a function of the input indices. For relational dependencies, DML offers existential types. These allow one to restrict the index of an output to a sort—because sorts can be defined in terms of already declared indices, relational dependencies can be expressed.

```
append : $\Pi\, n_1, n_2 : \mathbb{N} \,.\, \alpha\, \mathsf{list}(n_1) \times \alpha\, \mathsf{list}(n_2) \rightarrow \alpha\, \mathsf{list}(n_1 + n_2)$
fun append lp =
 case lp of
    (nil,l2) => l2
  | (cons(x,xs),l2) => cons(x,append(xs,l2))


flatten : $\Pi\, m, n : \mathbb{N} \,.\, \alpha\, \mathsf{llist}(m,n) \rightarrow \alpha\, \mathsf{list}(n)$
fun flatten ll =
 case ll of
    lnil => nil
  | lcons(xs,rest) => append(xs,flatten rest)

occurs : $\Pi\, h, s : \mathbb{N} \,.\, \mathsf{string} \times \mathsf{string}\ \mathsf{HBtree}(h,s) \rightarrow \mathsf{bool}$
fun occurs(e,t) =
 case t of
    Leaf => false
  | Node(t1,e',t2) => if e = e'
                      then true
                      else if e < e'
                           then occurs(s,t1)
                           else occurs(s,t2)
```

Figure 1: Some functions with their DML types

Consider, for example, a function that inserts a string into a balanced tree. Depending on how the tree is rebalanced, the result can be a tree of equal height or a tree higher by one. Similarly, the number of elements in the tree stays equal if the element to be inserted already was in the tree, otherwise the number is increased by one. A valid DML type for a correct `insert` function on height-balanced trees is as follows:

$$\begin{aligned}\Pi\, h, s : \mathbb{N} \,.\, & \mathsf{string} \times \mathsf{HBtree}(h,s) \rightarrow \\ & \exists\, h' : \{k : \mathbb{N} \mid h \leq k \leq h+1\} \,. \\ & \exists\, s' : \{k : \mathbb{N} \mid s \leq k \leq s+1\} \,. \\ & \mathsf{HBtree}(h',s')\end{aligned}$$

The type of the output tree restricts height and size to be either equal or larger by one than the height and size of the input tree, respectively.

2.2 A formal specification of DML

In the theoretical development of DML [12], three languages are defined, whose interplay is displayed graphically in Figure 2.

- ML_0 basically is an extension of Mini-ML with general pattern matching. It formalizes a manageable subset of ML.

$$\begin{tikzcd}
e^* : \tau \arrow[r, "\|\cdot\|"] \arrow[d, "\text{eval}"'] & e : \sigma \arrow[d, "\text{eval}"'] & e' : \tau \arrow[l, "|\cdot|"'] \arrow[ll, bend right, "\text{type elaboration}"'] \\
v^* : \tau \arrow[r, "\|\cdot\|"] & v : \sigma & \\
\underbrace{\hphantom{xxxx}}_{\mathrm{DML}_0^{\Pi}(C)} & \underbrace{\hphantom{xxxx}}_{\mathrm{ML}_0} & \underbrace{\hphantom{xxxx}}_{\mathrm{DML}_0(C)}
\end{tikzcd}$$

Figure 2: Interplay of languages

- $\mathrm{DML}_0^{\Pi}(C)$ is an explicitly typed language with dependent types, i.e., types that are indexed with elements from a constraint domain C. Its syntax is that of ML_0, adding abstraction over index variables and application of an expression to index expressions. A canonical erasure $\|\cdot\|$ that removes index-related syntax both from the term and the type language, maps $\mathrm{DML}_0^{\Pi}(C)$ into ML_0. The erasure commutes with evaluation.

- $\mathrm{DML}_0(C)$ enriches ML_0 with dependent types; it has the same type language as $\mathrm{DML}_0^{\Pi}(C)$. $\mathrm{DML}_0(C)$ requires type-annotations only for recursive definitions. An erasure on the type language extends to an erasure $|\cdot|$ that maps $\mathrm{DML}_0(C)$ into ML_0. A type-elaboration algorithm maps a $\mathrm{DML}_0(C)$ program with correct type annotations into $\mathrm{DML}_0^{\Pi}(C)$ such that (1) the type annotations are preserved and (2) the erasure of both terms results in the same ML_0 term.

$\mathrm{DML}_0^{\Pi}(C)$ allows easy type-checking, because it is explicitly typed and indices are part of the term language. For the same reason, however, $\mathrm{DML}_0^{\Pi}(C)$ is impractical for actual programming. Instead, the user works with $\mathrm{DML}_0(C)$, which corresponds to the language, the example programs of Section 2.1.2 are given in: Their displayed DML-types are the type-annotations that are required for the implicit recursive definitions. Type-checking is carried out by a type elaboration algorithm [12, Chapter 4], evaluation by applying the erasure and the ML_0 evaluation mechanism.

In the following, we first give some basic facts about constraint domains and the constraint language used to express the index objects for DML types. We then briefly describe $\mathrm{DML}_0^{\Pi}(C)$.[2] The description glosses over many details—we refer the reader to Xi's PhD thesis [12] for the complete picture.

[2]For simplicity, we restrict the presentation to the monomorphic case without existential types—polymorphism and existential types are treated in extensions of $\mathrm{DML}_0^{\Pi}(C)$.

2.2.1 Constraints in DML

A constraint domain C is defined by (1) a signature Σ that declares a base sort along with basic operations and predicates and (2) a Σ-structure. For example, for the constraint domain $\mathbb{Z}$, the signature declares the base sort $\mathbb{Z}$ and the usual operations ($+$, $-$, mod, etc.) and predicates ($<$, $\geq$, etc.) over integers; the Σ-structure is given by the standard model of integers.

$$
\begin{array}{lrcl}
\text{sorts} & \gamma & ::= & b \mid \mathbf{1} \mid \gamma_1 \times \gamma_2 \mid \{a : \gamma \mid P\} \\
\text{propositions} & P & ::= & \top \mid \bot \mid i = j \mid p(i) \mid P_1 \wedge P_2 \mid P_1 \vee P_2 \\
\text{objects} & i, j & ::= & a \mid f(i) \mid \langle\rangle \mid \langle i, j \rangle \mid \mathbf{fst}(i) \mid \mathbf{snd}(i) \\
\text{contexts} & \phi & ::= & \cdot \mid \phi, a : \gamma \mid \phi, P
\end{array}
$$

Figure 3: Constraint language

DML uses the constraint language defined in Figure 3 to express the index objects for DML types. New sorts can be defined by pairing already defined sorts or restricting an already defined sort with a sort proposition. Sort propositions are built from the basic predicates p of the constraint domain. Index sorts serve as types for index objects, in which basic operations f of the constraint domain can appear. An index context is given as a collection of index propositions and type declarations for index variables.

DML type-checking requires a constraint solver that is able to handle constraints of the form

$$\Phi \quad ::= \quad \top \mid \bot \mid i = j \mid p(i) \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid \forall a : \gamma.\Phi \mid \exists a : \gamma.\Phi$$

Constraint satisfaction under a given index context, which is written as $\phi \models \Phi$, is defined in the canonical way.

2.2.2 The language $\mathbf{DML}_0^{\Pi}(C)$

A grammar of the $\text{DML}_0^{\Pi}(C)$ syntax is given in Figure 4.

$$
\begin{array}{rcl}
\tau & ::= & \delta(i) \mid \mathbf{1} \mid (\tau_1 \times \tau_2) \mid (\tau_1 \to \tau_2) \mid \Pi\, a : \gamma \,.\, \tau \\
e & ::= & x \mid \langle\rangle \mid \langle e_1, e_2 \rangle \mid c[i_1] \ldots [i_n] \mid c[i_1] \ldots [i_n](e) \\
 & & \mid (\mathbf{case}\ e\ \mathbf{of}\ ms) \mid (\mathbf{lam}\ x : \tau \,.\, e) \mid e_1(e_2) \\
 & & \mid \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} \mid (\mathbf{fix}\ x : \tau . e) \\
 & & \mid (\lambda a : \gamma \,.\, e) \mid e[i] \\
p & ::= & x \mid c[a_1] \ldots [a_n] \mid c[a_1] \ldots [a_n](p) \mid \langle\rangle \mid \langle p_1, p_2 \rangle \\
ms & ::= & p \Rightarrow e \mid p \Rightarrow e \mid ms
\end{array}
$$

Figure 4: Syntax of $\text{DML}_0^{\Pi}(C)$

In the grammar of the type language, $\delta(i)$ stands for a data type δ that is indexed with index object i. The DML data-type declaration of an enriched data type $\delta(i)$ yields constructor types of form $\Pi\, a_1 : \gamma_1 \ldots \Pi\, a_k : \gamma_k\,.\, \tau \to \delta(i)$ for constructors without argument such as *nil*, and $\Pi\, a_1 : \gamma_1 \ldots \Pi\, a_k : \gamma_k\,.\, \tau \to \delta(i)$ for constructors with argument such as *cons*. Several examples of types appeared in Section 2.1.2.

In addition to the usual constructs, the term language provides abstraction over index variables ($\lambda a : \gamma\,.\, e$) and application of an expression to an index object ($e[i]$). Furthermore, a constructor c of a recursive data type only appears with a number of index arguments—index variables when appearing in a pattern p and index objects otherwise. The number and sorts of index arguments is determined by the constructor type, which is inferred from the corresponding data type definition (see Section 2.1.2).

A typing judgment for $\mathrm{DML}_0^{\Pi}(C)$ has the form

$$\phi; \Gamma \vdash e : \tau,$$

where ϕ is an index context and Γ a normal context; an overview over the typing rules for $\mathrm{DML}_0^{\Pi}(C)$ is presented in Appendix A.1.

Substitutions play a central role in the formalization of DML: They are used both in the definition of the typing system and the semantics (Appendix A.2). A substitution can both affect index variables and normal variables:

$$\theta ::= [\,] \mid \theta[a \mapsto i] \mid \theta[x \mapsto e]$$

For a substitution θ, its restriction to index variables is referred to as θ_ϕ, its restriction to normal variables as θ_Γ.

The application of a substitution θ to a term t is written $t[\theta]$. With $\theta_1 \circ \theta_2$ we denote the substitution mapping t to $(t[\theta_1])[\theta_2]$; with $\theta_1\theta_2$ we denote the substitution that behaves like θ_1 on all variables in $\mathbf{dom}(\theta_1)\backslash\mathbf{dom}(\theta_2)$, and like θ_2 on all other variables.

3 Extracting cost recurrences

We first give an intuitive account of our method for extracting cost recurrences from DML programs, deferring a formal treatment to Section 4. We then present examples illustrating some distinctive features of the method.

3.1 The intuition behind extracting cost recurrences

We extract cost recurrences from first-order DML programs of the form

```
F₁:$\Pi\, a_0 : \gamma_0 \ldots \Pi\, a_{j_1} : \gamma_{j_1}\,.\,(\rho_{10} \times \rho_{11} \cdots \times \rho_{1l_1}) \to \rho_1$
fun F₁(x₀,x₁,...,x$_{l_1}$) = e₁
   ⋮
F$_k$:$\Pi\, a_0 : \gamma_0 \ldots \Pi\, a_{j_k} : \gamma_{j_k}\,.\,(\rho_{k0} \times \rho_{k1} \cdots \times \rho_{kl_k}) \to \rho_k$
fun F$_k$(x₀,x₁,...,x$_{l_k}$) = e$_k$
```

where we write $\rho ::= \delta(i) \mid \mathbf{1} \mid (\rho_1 \times \rho_2)$ for first-order types; also data-type constructors are only allowed to take first-order arguments. Because indices are used to abstract from data to data size, we require that (1) all sorts γ have been constructed only with subsorts of $\mathbb{N}$ and (2) data types are enriched such that for any i, all indices appearing in a branch of a data type $\delta(i)$ must be bounded.[3]

We count cost in terms of the number of calls to user-defined functions F and to constructors c, assigning a cost of $\mathbf{c}_F$ and $\mathbf{c}_c$ for each call, respectively. $\mathbf{c}_F$ and $\mathbf{c}_c$ are constants of the domain in which cost is measured, e.g., the natural numbers.

The first step of extracting a cost recurrence from a DML program is type elaboration, which yields a $\mathrm{DML}_0^\Pi(\mathbb{Z})$ program.[4]

Example *For the* `append` *function from Figure 1, type elaboration yields the $DML_0^\Pi(\mathbb{Z})$ program displayed in Figure 5. Type elaboration makes the indices explicit in the term language: index variables n_1 and n_2 are abstracted over; pattern matching against cons introduces a new index variable n_1'; cons and the recursive call of append are passed index objects that describe the length of the respective list arguments passed to cons and append. Because $DML_0^\Pi(\mathbb{Z})$ is monomorphic, assume that the data type* list *has been defined for a fixed type of elements, say* string*. The constructors nil and cons then are typed as follows:*

$$\begin{array}{lcl} \mathit{nil} & : & \mathsf{list}(0) \\ \mathit{cons} & : & \Pi\, n : \mathbb{N}\,.\, \mathsf{string} \times \mathsf{list}(n) \to \mathsf{list}(n+1) \end{array}$$

$$\begin{array}{l}
\mathbf{fix}\ \mathit{append} : \Pi\, n_1 : \mathbb{N}\,.\,\Pi\, n_2 : \mathbb{N}\,.\,\mathsf{list}(n_1) \times \mathsf{list}(n_2) \\
\qquad\qquad\qquad\qquad\qquad\qquad \to \mathsf{list}(n_1 + n_2). \\
\quad \lambda n_1 : \mathbb{N}\,.\,\lambda n_2 : \mathbb{N}\,.\,\mathbf{lam}\ \mathit{lp} : \mathsf{list}(n_1) \times \mathsf{list}(n_2)\,. \\
\qquad \mathbf{case}\ \mathit{lp}\ \mathbf{of} \\
\qquad\quad \langle \mathit{nil}, l_2 \rangle \Rightarrow l_2 \\
\qquad\quad \mid \langle \mathit{cons}[n_1']\langle x, \mathit{xs}\rangle, l_2 \rangle \Rightarrow \\
\qquad\qquad\quad \mathit{cons}[n_1' + n_2]\langle x, \mathit{append}[n_1'][n_2]\langle \mathit{xs}, l_2 \rangle\rangle
\end{array}$$

Figure 5: The `append` function in $\mathrm{DML}_0^\Pi(C)$

We now describe intuitively how the extraction algorithm works. Note that all steps can be carried out automatically; for manipulating constraints, the algorithm uses a constraint solver for $\mathbb{Z}$.

[3] More precisely speaking, every constructor type $\Pi\, a_1 : \gamma_1 \ldots \Pi\, a_k : \gamma_k\,.\,\rho \to \delta(i)$ must be such that for a fixed $\delta(i)$, there are only finitely many $z_1, \ldots z_k$ such that $c[z_1]\ldots[z_k]$ is of type $\rho \to \delta(i)$. This condition is met for every data-type definition in which the indices convey size information: structures of a given size cannot contain substructures of arbitrary size.

[4] Because type elaboration as defined by Xi [12, Chapter 4] works on $\mathrm{DML}_0(C)$ programs, the program has to be rewritten in $\mathrm{DML}_0(\mathbb{Z})$. This is easily done by replacing the ML function-definition syntax with a recursive definition (keyword **fix**), a lambda-abstraction (keyword **lam**), and a case expression with a single pattern, and declaring $F_1 \ldots F_k$ in a row of nested let-statements.

The type of each function determines the arguments of the corresponding recurrence equation. A function

$$F : \Pi\, a_0 : \gamma_0 \ldots \Pi\, a_l : \gamma_l \,.\, \rho_1 \rightarrow \rho_2$$

gives rise to a recurrence equation F^c with $a_0 \ldots a_l$ as formal parameters.

Example (cont.) *For append, a recurrence equation append^c with formal parameters n_1 and n_2 is extracted.*

The extraction algorithm works on the body of the function definitions. The issues that have to be dealt with are

1. How to treat case expressions?
2. How to treat index variables introduced by pattern matching?
3. How to assign and add up cost?

How to treat case expressions? To abstract from data to data size, we need to turn case expressions, which examine data, into conditionals that examine data size. Such a transformation can be achieved using information contained in the $\mathrm{DML}_0^{\Pi}(\mathbb{Z})$ type derivation: During type checking, constraints over the index objects in the program are collected in an index context. Consider a branch of a case expression over some type ρ. The type derivation contains a collection of constraints that have to be satisfied when entering the branch, i.e., when the pattern of the branch is matched. By projecting out these constraints over the index variables contained in ρ, i.e., eliminating all other index variables, a guard for the corresponding branch of a conditional can be derived.

Example (cont.) *The case expression in append is type-checked under the index context*

$$\phi = n_1 : \mathbb{N}, n_2 : \mathbb{N}$$

For the two branches, additional constraints are generated:

- *For the branch with pattern $\langle nil, l_2\rangle$, the index context $n_1 = 0$ is generated.*
- *For the branch with pattern $\langle cons[n_1']\langle x, xs\rangle, l_2\rangle$, the index context $n_1' : \mathbb{N}, n_1 = n_1' + 1$ is generated.*

From the conjunction of ϕ and the newly generated index context of each branch, we can derive a condition in terms of n_1 and n_2 by projecting out over n_1 and n_2: The result is $n_1 = 0$ for the first branch and $n_1 > 0$ for the second branch.

In general, it is possible that the generated guards overlap, even though the patterns of the case expression are mutually exclusive. When, during the evaluation of a recurrence equation, more than one guard is satisfied, all possible branches are evaluated and the maximum value is returned.

How to treat index variables introduced by pattern matching? A pattern can introduce new index variables; these index variables may appear within the branch guarded by the pattern, and thus also may play a role in the corresponding conditional branch of the extracted recurrence equation. Often, we can eliminate such index variables by deriving equality constraints that define a new index variable in terms of other index variables. If not, then the constraints can be used to derive a restriction for the values of the new index variables. This restriction is inserted into the extracted conditional branch.

Example (cont.) *The second branch of the case expression in append introduces the new index variable* n_1'*. The constraints allow us to derive that* $n_1' = n_1 - 1$*, so* n_1' *can be eliminated.*

How to count and add up cost? When extracting a cost recurrence, we need to count every call to a user-defined function F with a cost of $\mathbf{c}_F$ and every use of a constructor c with a cost of $\mathbf{c}_c$. Consider first a constructor c without arguments: In the cost recurrence, we simply replace $c[i_1]\dots[i_k]$ with $\mathbf{c}_c$. For constructors with arguments $c[i_1]\dots[i_k](e)$ and function calls $F[i_1]\dots[i_k](e)$, the cost incurred by e also needs to be taken into account. Hence, we first extract a recurrence-equation expression t that represents the cost of evaluating the argument, and then add it to the cost incurred by the function call:

- The total cost of $c[i_1]\dots[i_k](e)$ is $t + \mathbf{c}_c$
- The total cost of $F[i_1]\dots[i_k](e)$ is $t + \mathbf{c}_F + (F^c i_1 \dots i_k)$, where $F^c i_1 \dots i_k$ is a call to the cost recurrence extracted for F.

In our cost model, constants and variables can be accessed without cost, and therefore are turned into the constant 0 when extracting a cost recurrence.

Example (ended) *We now assemble all the pieces of a cost recurrence for append. If we assign a cost of one unit to recursive calls of append and assume the use of cons to be cost free, then the cost of append is described by*

$$\mathit{append}^c\ n_1\ n_2 = \begin{cases} n_1 = 0 \to 0 \\ n_1 > 0 \to 1 + \mathit{append}^c (n_1 - 1)\ n_2 \end{cases}$$

(In the second branch, we have removed additions of zero that resulted from the variables x*,* xs *and* l_2*, and the application of constructor cons.)*

3.2 Example: Flattening a list of lists

The `flatten` function (see Figure 1) is an interesting problem for extracting a cost recurrence because of the choice of size measure for the input: The size of a list of lists is measured both in terms of the number of inner lists and the total number of elements contained in the inner lists.

We measure cost in terms of calls to user-defined functions. Our method yields the following cost recurrence:[5]

$$\mathit{flatten}^c\ m\ n = \begin{cases} m = 0 \wedge n = 0 \rightarrow 0 \\ m > 0 \rightarrow 2 + \mathit{append}^c\ n_1\ n_2 \\ \qquad\qquad + \mathit{flatten}^c\ (m-1)\ n_2 \end{cases}$$
$$\text{where} \quad n_1 + n_2 = n$$

Here, a restriction $n_1 + n_2 = n$ for the new variables n_1 and n_2 introduced by pattern matching has been inserted by the extraction algorithm—neither n_1 nor n_2 can be eliminated automatically.

Using the equation $\mathit{append}^c\ n_1\ n_2 = n_1$ derived in Section 3.1, we can rewrite the cost recurrence for *flatten* as

$$\mathit{flatten}^c\ m\ n = \begin{cases} m = 0 \wedge n = 0 \rightarrow 0 \\ m > 0 \rightarrow 2 + n_1 \\ \qquad\qquad + \mathit{flatten}^c\ (m-1)\ n_2 \end{cases}$$
$$\text{where} \quad n_1 + n_2 = n$$

It is easy to see that the maximal cost incurred by n_1 in the second branch, added over all recursive calls, is n; all in all, we can approximate the cost of `flatten` as $\mathit{flatten}^c(m, n) = 2m + n$.

The size measure chosen here for a list of lists is intuitive and seems to be crucial for deriving a useful bound. Yet it is unclear how this measure could be be defined without the expressiveness offered by DML types, e.g., when using abstract-interpretation techniques [8].

3.3 Example: Searching a balanced tree

The `occurs` function (see Figure 1) provides an example of how two cost bounds in terms of different size measures can be obtained: one in terms of the height of a tree, and one in terms of the number of elements stored in a tree. The latter bound is obtained by reasoning with DML type guarantees—we profit from the fact that DML can express properties of the input that are not inferable from the code.

Our method yields the following cost recurrence for `occurs`:

$$\mathit{occurs}^c\ h\ s = \begin{cases} h = 0 \wedge s = 0 \rightarrow 0 \\ h > 0 \wedge s > 0 \rightarrow \begin{cases} 0 \\ \begin{cases} 1 + \mathit{occurs}^c\ h_1\ s_1 \\ 1 + \mathit{occurs}^c\ h_2\ s_2 \end{cases} \end{cases} \end{cases}$$
$$\begin{aligned} \text{where} \quad & h_1 - 1 \leq h_2 \leq h_1 + 1 \\ & \wedge\ \mathit{max}(h_1, h_2) + 1 = h \\ & \wedge\ s_1 + s_2 + 1 = s \end{aligned}$$

[5]Here and in all the following recurrences we have simplified additions of constants.

(Each if expression gives rise to a guardless conditional, because no restrictions on indices can be inferred from its test expression.)

The recurrence looks more daunting than it is: It keeps track both of the height and the number of elements in the tree, but it is easy to see that the number of elements is of no consequence to the result of the cost recurrence. Approximating both h_1 and h_2 with $h - 1$ gives rise to a simple recurrence equation whose solution is $\mathit{occurs}^c(h, s) = h$.

The complication of eliminating size information could have been avoided by chosing a tree type which only keeps track of the height of a tree. Also keeping track of the number of elements, however, allows us to derive a cost measure in terms of the number of elements rather than the height of the tree. The crucial observation to make is that DML data-type definitions give rise to induction principles for proving relations among the indices of a data type. The definition of HBtree, for example, yields the following induction schema:

For $R \in \mathbb{N} \times \mathbb{N}$, if

1. $R(0, 0)$
2. if for all h_1, h_2, s_1, s_2 with $h_1 - 1 \leq h_2 \leq h_1 + 1$, $R(h_1, s_1)$ and $R(h_2, s_2)$ it follows that $R(\mathit{max}(h_1, h_2) + 1, s_1 + s_2 + 1)$

then whenever a value has type $\mathsf{HBtree}(h,s)$, the relation $R(h, s)$ holds.

Using this induction schema, one can show that $s \geq 2^h - 1$ for any height-balanced tree with height h and size s. Taking the logarithm, we see that $h \leq \mathit{log}(s + 1)$; combining this with the cost recurrence $\mathit{occurs}^c(h, s) = h$, we derive that the cost of `occurs` is logarithmic in s. This derivation is fully formal, i.e., based only on assumptions explicit in the types or the extracted recurrence.

3.4 Example: Merge sort

Merge sort provides an example of how the extraction algorithm preserves useful information contained in a program's DML type.

An implementation of merge sort in DML is given in Figure 6 (adapted from the distribution of *de Caml*, a DML prototype [11]): Function `initlist` converts the list to be sorted into a list of lists such that each of these lists is sorted and has length two (apart from a possible last singleton list). Function `merge2` goes through a list of lists, merging every two adjacent lists into one. Function `mergeall` iterates the application of `merge2` until a single list is obtained. The types of `initlist` and `merge2` capture the fact that the size measure that steers the recursion is continually halved—the index expression $\lceil n/2 \rceil$ can be encoded as $\mathit{div}(n, 2) + \mathit{mod}(n, 2)$ in the integer constraint domain we are working with.

We extract cost recurrences (Figure 7a), this time counting the number of comparisons (counting calls to primitive functions works the same as counting calls to user-defined functions). The recurrences for merge^c, $\mathit{initlist}^c$ and $\mathit{merge2}^c(m, n)$ are easy to approximate, yielding a simplified set of recurrences merge : $\Pi\, n_1, n_2 : \mathbb{N} .\, \mathsf{list}(n_1) \times \mathsf{list}(n_2) \rightarrow \mathsf{list}(n_1 + n_2)$

```
fun merge lp =
 case lp of
    (nil, l2) => l2
  | (l1, nil) => l1
  | (cons(h1,t1),cons(h2,t2)) =>
      if h1 < h2 then cons(h1,merge(t1,l2))
                 else cons(h2,merge(l1,t2))
```

initlist : $\Pi\, n : \mathbb{N} .\, \mathsf{list}(n) \rightarrow \mathsf{llist}(\lceil n/2 \rceil, n)$

```
fun initlist l =
 case l of
    nil => lnil
  | cons(_,nil) => lcons(l, lnil)
  | cons(e1,cons(e2, rest)) =>
      lcons(if e1 < e2
            then cons(e1,cons(e2,nil))
            else cons(e2,cons(e1,nil)),
            initlist rest)
```

merge2 : $\Pi\, m, n : \mathbb{N} .\, \mathsf{llist}(m, n) \rightarrow \mathsf{llist}(\lceil m/2 \rceil, n)$

```
fun merge2 ll =
 case ll of
    lnil => ll
  | lcons(_,lnil) => ll
  | lcons(l1,lcons(l2,rest)) =>
       lcons(merge(l1,l2),merge2 rest)
```

mergeall : $\Pi\, m, n : \mathbb{N} .\, \mathsf{llist}(m, n) \rightarrow \mathsf{list}(n)$

```
mergeall ll =
 case ll of
    lnil => nil
  | lcons(l,lnil) => l
  | lcons(_,lcons(_,_)) => mergeall(merge2 ll)
```

msort : $\Pi\, n : \mathbb{N} .\, \mathsf{list}(n) \rightarrow \mathsf{list}(n)$

```
msort l = mergeall(initlist l)
```

Figure 6: Merge sort in DML

displayed in Figure 7b. Notice how the information about halving the length of the list of lists captured in the type of `merge2` appears in $mergeall^c\ m\ n$. The solution of this recurrence equation is well-known to be $\mathcal{O}(n\ log\ m)$, which gives an overall complexity for `msort` of $\mathcal{O}(n\ log\ n)$.

An extraction scheme without access to such high-level information as provided by DML types might still provide enough *implicit* information in the

$$\mathit{merge}^c\ n_1\ n_2 = \begin{cases} n_1 = 0 \to 0 \\ n_2 = 0 \to 0 \\ n_1 > 0 \wedge n_2 > 0 \to 1 + \begin{cases} \mathit{merge}^c\ (n_1 - 1)\ n_2 \\ \mathit{merge}^c\ n_1\ (n_2 - 1) \end{cases} \end{cases}$$

$$\mathit{initlist}^c\ n = \begin{cases} n = 0 \to 0 \\ n = 1 \to 0 \\ n > 1 \to 1 + \mathit{initlist}^c\ (n-2) \end{cases}$$

$$\mathit{merge2}^c\ m\ n = \begin{cases} m = 0 \wedge n = 0 \to 0 \\ m = 1 \to 0 \\ m > 1 \to \mathit{merge}^c\ n_1\ n_2 + \mathit{merge2}^c\ (m-2)\ n_3 \\ \quad \text{where} \quad n_1 + n_2 + n_3 = n \end{cases}$$

$$\mathit{mergeall}^c\ m\ n = \begin{cases} m = 0 \wedge n = 0 \to 0 \\ m = 1 \to 0 \\ m > 1 \to \mathit{merge2}^c\ m\ n + \mathit{mergeall}^c(\lceil m/2 \rceil)\ n \end{cases}$$

$$\mathit{msort}^c\ n = \mathit{initlist}^c\ n + \mathit{mergeall}^c\ (\lceil n/2 \rceil)\ n$$

a: Extracted cost recurrences

$$\mathit{merge}^c\ n_1\ n_2 = n_1 + n_2$$

$$\mathit{initlist}^c\ n = \lfloor n/2 \rfloor$$

$$\mathit{merge2}^c\ m\ n = n$$

$$\mathit{mergeall}^c\ m\ n = \begin{cases} m \leq 1 \to 0 \\ m > 1 \to n + \mathit{mergeall}^c\ (\lceil m/2 \rceil)\ n \end{cases}$$

$$\mathit{msort}^c\ n = \lfloor n/2 \rfloor + \mathit{mergeall}^c\ (\lceil n/2 \rceil)\ n$$

b: Approximated cost recurrences

Figure 7: Cost recurrences for merge sort

extracted cost bound to derive the same bound, but the reasoning over the cost recurrence would be more involved. Basically, information about argument sizes that is encoded in the DML type and carried over into the cost recurrence with our method, would first have to be (re)proven for the cost bound.

4 Formal development

We now formally define the method for extracting cost recurrences from DML programs. The development is based on the theoretical view of DML presented in Section 2.2 and therefore only treats a monomorphic version of DML without existential types. Extending the development into a polymorphic setting is straightforward and has been omitted for the sake of conciseness. For simplicity, the formalization also does not treat mutual recursion. Extracting a cost bound from mutually recursive programs works exactly the same, but checking whether an extracted bound is indeed a recurrence becomes somewhat more tricky. For the latter, techniques such as presented in Xi's latest work [14] could be used (see Section 5).

A first assessment shows that our method can also be extended to existential types in a straightforward way. Essentially, the application of a function that returns a value of existential type introduces a new index variable that can be treated in the same way as new index variables introduced by pattern matching.

We start by defining the first-order fragment of DML treated by our method. For this fragment, we introduce a cost measure using a monadic translation with a cost monad. After defining a language of recurrence equations, we present the extraction algorithm. We prove its correctness by showing that extraction, if successful, indeed yields an upper bound with respect to the cost model defined by the monadic translation.

4.1 A first-order fragment of DML

As pointed out in Section 3.1, the first step of extracting cost recurrences from a first-order $\mathrm{DML}_0(\mathbb{Z})$ program is type-elaboration, which results in a $\mathrm{DML}_0^{\Pi}(\mathbb{Z})$ program of the form given in Figure 8 (we abbreviate a row of abstractions over $a_0 : \gamma_0, \ldots a_l : \gamma_l$ with $\lambda \vec{a} : \vec{\gamma}$, and $\langle x_0, \langle x_1, \ldots, \langle x_{i-1}, x_i \rangle \rangle \rangle$ with $\langle x_0, \ldots, x_i \rangle$). The extraction algorithm to be presented in Section 4.4 therefore operates on the language defined in Figure 8 (cf. the full language in Figure 4).

The original semantics of DML (see Appendix A.2) is defined as a natural semantics: $e \longrightarrow v$ means that e evaluates under environment Θ to value v. The semantics has a rule ev-fix for unfolding fixed-point definitions to handle recursion. For the first-order fragment of DML with its restricted form of function definition and function application used here, it is convenient to define a semantics that handles recursion using an environment of function definitions. We define a modified semantics: $e \longrightarrow_{\Theta} v$ means that e evaluates under environment Θ to value v, where Θ is a substitution mapping function names to their definitions. We show the following theorem:

Theorem 1
Let p be a program of the form given in Figure 8. Then $p \longrightarrow v$ in the standard semantics iff $p \longrightarrow_{[]} v$ in the modified semantics.

The definition of the modified semantics and the proof of Theorem 1 are deferred to Appendix B.1.

$$
\begin{array}{l}
\textbf{let } F_1 = \textbf{fix } F_1 : \Pi\, a_0 : \gamma_0 \ldots \Pi\, a_{l_1} : \gamma_{l_1} \,.\, \rho_{11} \to \rho_{12} . \\
\qquad\qquad \lambda \vec{a} : \vec{\gamma} \,.\, \textbf{lam } x : \rho_{11} \,.\, \mathit{body} \\
\vdots \\
\quad F_k = \textbf{fix } F_k : \Pi\, a_0 : \gamma_0 \ldots \Pi\, a_{l_k} : \gamma_{l_k} \,.\, \rho_{k1} \to \rho_{k2} . \\
\qquad\qquad \lambda \vec{a} : \vec{\gamma} \,.\, \textbf{lam } x : \rho_{k1} \,.\, \mathit{body} \\
\textbf{in } e \\
\textbf{end}
\end{array}
$$

a: Def. of functions $F_1 \ldots F_k$ in $\mathrm{DML}_0^{\Pi}(\mathbb{Z})$

$$
\begin{array}{rcl}
\mathit{body} & ::= & \textbf{case } x \textbf{ of } \langle x_0, \ldots, x_{i_k} \rangle \Rightarrow e \\
e & ::= & x \mid \langle\rangle \mid \langle e_1, e_2 \rangle \mid c[i_1] \ldots [i_n] \mid c[i_1] \ldots [i_n](e) \\
 & & \mid (\textbf{case } e \textbf{ of } ms) \mid \textbf{let } x = e_1 \textbf{ in } e_2 \textbf{ end} \\
 & & \mid F_i[i_1] \ldots [i_n](e) \\
p & ::= & x \mid c[a_1] \ldots [a_n] \mid c[a_1] \ldots [a_n](p) \mid \langle\rangle \mid \langle p_1, p_2 \rangle \\
ms & ::= & p \Rightarrow e \mid p \Rightarrow e \mid ms
\end{array}
$$

b: Grammar of function bodies

Figure 8: A first-order fragment of $\mathrm{DML}_0^{\Pi}(\mathbb{Z})$

4.2 Measuring cost of computation

One way of introducing a cost measure into functional programs is the monadic translation [6] with a cost monad. It is well-known that state can be added to a program by (1) performing a monadic translation with the state monad [10] and (2) taking the term model of the result, i.e., expanding the monadic constructs inserted by the translation to code. Similarly, using the cost monad instead of the state monad, we can transform a program such that a cost counter is maintained.

The cost monad pairs computations that result in a value of type τ with a second component of a type $\mathbf{C}$ that expresses the cost of the computation; all we need to know is that $\mathbf{C}$ is an ordered Abelian monoid[6] $(\mathbf{C}, +, \mathbf{0}, \leq)$. We write $\mathsf{C}\,\alpha$ as a type abbreviation for $\alpha \times \mathbf{C}$. A call-by-value monadic translation with a cost monad that is based on C turns a function of type $\Pi\, a_0 : \gamma_0 \ldots \Pi\, a_k : \gamma_k \,.\, \rho_1 \to \rho_2$ into a function of type $\Pi\, a_0 : \gamma_0 \ldots \Pi\, a_k : \gamma_k \,.\, \rho_1 \to \mathsf{C}\,\rho_2$. The intended meaning is that the transformed function not only returns the result value, but also the cost of computing it.

The cost monad can be defined by specifying two language constructs, $\mathbf{val}^{\mathsf{C}}$ and $\mathbf{let}^{\mathsf{C}}$, which a monadic translation inserts into a program text. The typing

[6]In an ordered monoid, the ordering $\leq$ is compatible with the monoid's operation, i.e., if $a \leq b$ then $a + c \leq b + c$. Relevant examples of ordered monoids are $(\mathbb{N}, +, 0, \leq)$ and $(\mathbb{R}, +, 0, \leq)$.

rule for $\mathbf{val}^{\mathsf{C}}$ is

$$\frac{\phi; \Gamma \vdash e : \rho}{\phi; \Gamma \vdash \mathbf{val}^{\mathsf{C}}\ e : \mathsf{C}\,\rho}\ (\textsf{ty-monadic-val})$$

The construct $\mathbf{val}^{\mathsf{C}}$ is used to inject a value $v : \rho$ into $\mathsf{C}\,\rho$ as $\langle v, \mathbf{0} \rangle$—values do not require any computation and thus incur no cost. The typing rule for $\mathbf{let}^{\mathsf{C}}$ is

$$\frac{\phi; \Gamma \vdash e_1 : \mathsf{C}\,\rho_1 \qquad \phi; \Gamma, x : \rho_1 \vdash e_2 : \mathsf{C}\,\rho_2}{\phi; \Gamma \vdash \mathbf{let}^{\mathsf{C}}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} : \mathsf{C}\,\rho_2}\ (\textsf{ty-monadic-let})$$

In ($\mathbf{let}^{\mathsf{C}}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end}$), the expression e_1 is evaluated to a result v_1 wrapped with a cost z_1. To calculate e_2, the unwrapped v_1 is used, yielding $\langle v_2, z_2 \rangle$. The final result of the let expression is $\langle v_2, z_1 + z_2 \rangle$.

The monadic translation provides only the infrastructure for tracking cost, but does not assign costs to any program constructs. This assignment of costs is done by inserting a monadic construct $\mathbf{cost}_z$ with typing rule

$$\frac{\phi; \Gamma \vdash e : \mathsf{C}\,\rho \qquad z \in \mathbf{C}}{\phi; \Gamma \vdash \mathbf{cost}_z\ e : \mathsf{C}\,\rho}\ (\textsf{ty-monadic-cost})$$

into the transformed program. $\mathbf{cost}_z$ is particular to the cost monad: It adds z to the cost component of the value it is applied to.

A *cost-conscious* version of a program, i.e., a program that keeps track of the cost incurred by calls to user-defined functions and uses of constructors, is generated as follows: We first perform a monadic translation of the program and then enclose (1) each application of a user-defined function F with $\mathbf{cost}_{\mathbf{c}_F}$, and (2) each application of a constructor c with $\mathbf{cost}_{\mathbf{c}_c}$.

Figure 9 displays the combined translation $(\cdot)^*$ for function bodies. A program p of the form given in Figure 8 is translated into p^* by applying the monadic translation to all function bodies and the body of the program, and changing every result type ρ_{l2} in the type annotation of the fixed-point definition to $\mathsf{C}\,\rho_{l2}$. The cost-conscious version can easily be expressed in $\mathrm{DML}_0^{\Pi}(C)$: Figure 10 shows how to expand $\mathbf{val}^{\mathsf{C}}$, $\mathbf{let}^{\mathsf{C}}$ and $\mathbf{cost}_z$.

The following theorem shows that the cost translation is well-behaved:

Theorem 2
Let $\cdot; \cdot \vdash \mathsf{p} : \rho$ be derivable. Then

1. *judgment $\cdot; \cdot \vdash \mathsf{p}^* : \mathsf{C}\,\rho$ is derivable.*
2. *$\mathsf{p} \longrightarrow_{[]} v$ iff $\mathsf{p}^* \longrightarrow_{[]} \langle v, z \rangle$ for some $z \in \mathbf{C}$.*

The proof is deferred to Appendix B.2.

4.3 A language of recurrence equations

The language of recurrence equations is based on the natural numbers part $\mathbb{N}$ of the constraint domain $\mathbb{Z}$ and the cost domain $\mathbf{C}$. Natural numbers and tuples thereof serve as abstractions of input size, and therefore are used as arguments of recurrence equations. The result of a recurrence equation represents cost of computation and is expressed in $\mathbf{C}$.

$$
\begin{array}{lcl}
x^* & = & \mathbf{val}^{\mathsf{C}}\ x \\
\langle\rangle^* & = & \mathbf{val}^{\mathsf{C}}\ \langle\rangle \\
\langle e_1, e_2\rangle^* & = & \mathbf{let}^{\mathsf{C}}\ x_1 = {e_1}^*\ \mathbf{in} \\
 & & \mathbf{let}^{\mathsf{C}}\ x_2 = {e_2}^* \\
 & & \mathbf{in\ val}^{\mathsf{C}}\ \langle x_1, x_2\rangle\ \mathbf{end} \\
(c[i_1]\dots[i_k])^* & = & \mathbf{cost}_{\mathbf{c}_c}\ (\mathbf{val}^{\mathsf{C}}\ (c[i_1]\dots[i_k])) \\
(c[i_1]\dots[i_k](e))^* & = & \mathbf{let}^{\mathsf{C}}\ x = e^* \\
 & & \mathbf{in\ cost}_{\mathbf{c}_c}\ (\mathbf{val}^{\mathsf{C}}\ (c[i_1]\dots[i_k](x)))\ \mathbf{end} \\
(\mathbf{case}\ e\ \mathbf{of}\ ms)^* & = & \mathbf{let}^{\mathsf{C}}\ x = e^* \\
 & & \mathbf{in\ case}\ x\ \mathbf{of}\ ms^*\ \mathbf{end} \\
(p \Rightarrow e \mid ms)^* & = & p \Rightarrow e^* \mid ms^* \\
(\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end})^* & = & \mathbf{let}^{\mathsf{C}}\ x = {e_1}^* \\
 & & \mathbf{in}\ {e_2}^*\ \mathbf{end} \\
(F[i_1]\dots[i_k](e))^* & = & \mathbf{let}^{\mathsf{C}}\ x = e^* \\
 & & \mathbf{in\ cost}_{\mathbf{c}_c}\ (F[i_1]\dots[i_k](x))\ \mathbf{end}
\end{array}
$$

Figure 9: Monadic translation of function bodies

$$
\begin{array}{lcl}
\mathbf{val}^{\mathsf{C}}\ e & \equiv & \langle e, \mathbf{0}\rangle \\
\mathbf{let}^{\mathsf{C}}\ x = e_1 & \equiv & \mathbf{case}\ e_1\ \mathbf{of} \\
\mathbf{in}\ e_2 & & \quad \langle x, z_1\rangle \Rightarrow\ \mathbf{case}\ e_2\ \mathbf{of} \\
\mathbf{end} & & \qquad\qquad \langle v, z_2\rangle \Rightarrow \langle v, z_1 + z_2\rangle \\
\mathbf{cost}_z\ e & \equiv & \mathbf{case}\ e\ \mathbf{of} \\
 & & \quad \langle x, z'\rangle \Rightarrow\ \langle x, z + z'\rangle
\end{array}
$$

Figure 10: Monadic constructs as syntactic sugar

4.3.1 Syntax and types

We describe a system of recurrence equations with the language given in Figure 11. Because we extract recurrences from programs that are not mutual recursive, neither is a system of recurrence equations, i.e., a body t_l may only contain recurrence-equation names $F_1^c \dots F_l^c$. Conditionals, which so far have been pretty printed, are introduced with the keyword **cond** followed by a number of branches. Within a branch, the first constraint Φ_1 represents the guard of the branch, whereas the second constraint Φ_2 represents a *where*-clause. The scope of the quantification (we write $\vec{a} : \vec{\sigma}$ as shorthand for the quantification over variables $a_1 : \sigma_1 \ \ldots a_k : \sigma_k$) extends both over Φ_2 and the branch body t. For $\forall\vec{a} : \vec{\sigma}.\Phi_2$ we require that for any interpretation of its free variables, there are only finitely many instantiations of $\vec{a}$ such that Φ_2 is satisfied; this requirement is met for recurrence equations extracted from DML programs in which the data types are enriched in a sensible way as required in Section 3.1.

$$
\begin{array}{llcl}
\text{index types} & \sigma & ::= & \mathbb{N} \mid \mathbf{1} \mid \sigma_1 \times \sigma_2 \\
\text{types} & \nu & ::= & \mathbf{C} \mid \sigma \to \nu \\
\text{definitions} & \mathfrak{E} & ::= & F_1^c\ a_0\ a_1 \ldots a_{l_1} = t_1 \\
 & & & \quad \vdots \\
 & & & F_k^c\ a_0\ a_1 \ldots a_{l_k} = t_k \\
\text{body} & t & ::= & z \mid t_1 + t_2 \mid F^c\ \vec{\imath} \mid (\mathbf{cond}\ brs) \\
\text{index objects} & i, j & ::= & a \mid f(i) \mid \langle\rangle \mid \langle i, j\rangle \mid \mathbf{fst}(i) \mid \mathbf{snd}(i) \\
\text{branches} & brs & ::= & br \mid br \mid brs \\
\text{branch} & br & ::= & (\Phi_1 \to \forall\vec{a} : \vec{\sigma}.\Phi_2 \to t)
\end{array}
$$

Figure 11: Syntax of recurrence equations

The rationale behind the shape of recurrence equation types is that for a function F_l of type

$$\Pi\, a_0 : \gamma_0 \,.\, \Pi\, a_1 : \gamma_1 \ldots \Pi\, a_k : \gamma_k \,.\, \rho_1 \to \rho_2,$$

the associated cost recurrence F_l^c should have type

$$\widetilde{\gamma_0} \to \widetilde{\gamma_1} \to \ldots \widetilde{\gamma_k} \to \mathbf{C},$$

where $\widetilde{\cdot}$ maps an index sort to the associated index type. For example $\{a : \mathbb{N} \times \mathbb{N} \mid \mathbf{fst}(a) \leq \mathbf{snd}(a)\}$ is mapped to $\mathbb{N} \times \mathbb{N}$.

The formal definition of $\widetilde{\cdot}$ and its extension to contexts Γ is deferred to Appendix B.3.1.

The body of a recurrence equation F_k^c is typed using the judgment

$$\varphi; \Delta \vdash e : \nu$$

in which φ maps index types σ to index variables, and Δ assigns recurrence-equation types to function names. Figure 12 gives typing rules, all of which are straightforward. So are the rules for checking the type of an index object ($\varphi \vdash i : \sigma$) and the wellformedness of a constraint ($\varphi \vdash \Phi$), which have been omitted.

Based on the typing rules for the body of a recurrence equation, we define what it means for a system of recurrence equations to be well-typed with respect to the program they have been extracted from.

Definition 3
Let p *be a program and context* Γ *assign every* F_l *defined in* p *to its declared type. A system of recurrence equations* $\mathfrak{E}$ *is well-typed with respect to* p *if for*

every

$$F : \Pi\, a_0 : \gamma_0 \,.\, \Pi\, a_1 : \gamma_1 \ldots \Pi\, a_k : \gamma_k \,.\, \rho_1 \to \rho_2$$

defined in `p` *, for the extracted recurrence equation*

$$F^c\ a_0 \ldots\ a_k \ =\ t$$

we have

$$a_0 : \widetilde{\gamma_0}, \ldots, a_k : \widetilde{\gamma_k}; \widetilde{\Gamma} \vdash t : \mathbf{C}.$$

$$\frac{}{\varphi; \Delta \vdash\ z : \mathbf{C}}\ \textsf{(ty-re-const)}$$

$$\frac{\varphi; \Delta \vdash\ e_1 : \mathbf{C} \qquad \varphi; \Delta \vdash\ e_2 : \mathbf{C}}{\varphi; \Delta \vdash\ e_1 + e_2 : \mathbf{C}}\ \textsf{(ty-re-plus)}$$

$$\frac{\begin{array}{c}\Delta(F) = \sigma_0 \to \ldots \to \sigma_k \to \mathbf{C} \\ \varphi \vdash i_0 : \sigma_0 \quad \ldots \quad \varphi \vdash i_k : \sigma_k\end{array}}{\varphi; \Delta \vdash\ F^c\ i_0 \ldots i_k : \mathbf{C}}\ \textsf{(ty-re-app)}$$

$$\frac{\begin{array}{c}\varphi \vdash \Phi_{11} \quad \varphi, \vec{a_1} : \vec{\sigma_1} \vdash \Phi_{12} \quad \varphi, \vec{a_1} : \vec{\sigma_1}; \Delta \vdash\ e_1 : \mathbf{C} \\ \vdots \\ \varphi \vdash \Phi_{k1} \quad \varphi, \vec{a_k} : \vec{\sigma_k} \vdash \Phi_{k2} \quad \varphi, \vec{a_k} : \vec{\sigma_k}; \Delta \vdash\ e_k : \mathbf{C}\end{array}}{\begin{array}{l}\varphi; \Delta \vdash (\mathbf{cond}\ \Phi_{11} \to \forall \vec{a_1} : \vec{\sigma_1}.\Phi_{12} \to e_1 \\ \qquad \vdots \\ \qquad\qquad \mid \Phi_{k1} \to \forall \vec{a_k} : \vec{\sigma_k}.\Phi_{k2} \to e_k) : \mathbf{C}\end{array}}\ \textsf{(ty-re-cond)}$$

Figure 12: Typing rules for recurrence equations

4.3.2 Semantics

We give a simple denotational semantics to the language of recurrence equations. A recurrence equation defining a function of type

$$\sigma_0 \to \sigma_1 \to \ldots \sigma_k \to \mathbf{C}$$

is interpreted in the function domain

$$[\mathcal{I}[\![\sigma_0]\!] \to \mathcal{I}[\![\sigma_1]\!] \to \ldots \mathcal{I}[\![\sigma_k]\!] \to \mathbf{C}_\perp].$$

(Because all $\mathcal{I}[\![\sigma_l]\!]$ are discrete cpos, such functions are necessarily continuous).

Here $\mathcal{I}[\![\cdot]\!]$ is the canonical semantics given to ground index objects i and index types σ by the constraint domain $\mathbb{Z}$; for an index substitution θ that maps index

$$
\begin{array}{rcl}
\mathcal{T}[\![z]\!]\Psi\theta & = & \llcorner z \lrcorner \\
\mathcal{T}[\![t_1 + t_2]\!]\Psi\theta & = & \mathcal{T}[\![t_1]\!]\Psi\theta +_\perp \mathcal{T}[\![t_2]\!]\Psi\theta \\
\mathcal{T}[\![F_l^c\ i_0 \dots i_k]\!]\Psi\theta & = & \Psi(F_l^c)\ (\mathcal{I}[\![i_0[\theta]]\!]) \dots (\mathcal{I}[\![i_k[\theta]]\!]) \\
\mathcal{T}[\![\mathbf{cond}\ br_0 \mid \dots \mid br_k]\!]\Psi\theta & = & max_\perp\{\mathcal{B}[\![br_0]\!]\Psi\theta, \dots, \mathcal{B}[\![br_k]\!]\Psi\theta\} \\
\mathcal{B}[\![(\Phi_1 \rightarrow \forall\vec{a} : \vec{\sigma}.\Phi_2 \rightarrow t)]\!]\Psi\theta & = & \begin{cases} \llcorner \mathbf{0} \lrcorner \quad \text{if } \not\models \Phi_1[\theta] \\ max_\perp\{\mathcal{T}[\![t]\!]\Psi\theta[\vec{a} \mapsto \vec{z}] \mid \\ \qquad \vec{z} \in \mathcal{I}[\![\vec{\sigma}]\!] \ \wedge \Phi_2[\theta[\vec{a} \mapsto \vec{z}]]\} \\ \qquad \text{if } \models \Phi_1[\theta] \end{cases}
\end{array}
$$

Figure 13: Semantics of recurrence equations

variables to ground index objects, we write $\mathcal{I}[\![\theta]\!]$ for the corresponding mapping into $\mathbb{Z}$. With $\mathbf{C}_\perp$ we denote the domain that results from interpreting $\mathbf{C}$ as a discrete domain and lifting it in the canonical way—in the semantics definition we also mark operations on $\mathbf{C}$ that have been lifted in the canonical way by subscripting them with $\perp$.

Figure 13 gives a semantics $\mathcal{T}[\![\cdot]\!]\Psi\theta$ for recurrence-equation expressions, where Ψ is a mapping from recurrence-equation names to functions and θ an index substitution that maps index variables to ground index objects. The semantics treats a conditional by taking the maximum of the values returned by the branches of the conditional. If the guard of a branch does not hold, the branch returns $\mathbf{0}$. Otherwise, all possible values for the universally quantified index variables in the branch are tried out and the maximum is returned. Because, as required in Section 4.3.1, there is at most a finite number of such values, the semantics of a branch is well-defined (we assume that $max_\perp\emptyset = \llcorner \mathbf{0} \lrcorner$).

Given the definition of a recurrence equation

$$F^c a_0\ a_1 \dots a_k = t$$

and a mapping Ψ that ranges over all names of recurrence-equations declared previously to F^c (recall that we do not consider mutually recursive systems of recurrence equations), then the semantics of the defined function is the fixed point of the functional

$$\lambda\mathcal{F}.\lambda n_0\ n_1 \dots n_k.\mathcal{T}[\![t]\!](\Psi[F^c \mapsto \mathcal{F}])[a_0, \dots, a_k \mapsto n_0, \dots n_k].$$

This semantics of a single recurrence equation extends naturally to a system $\mathfrak{E}$ of recurrence equations and yields a mapping from recurrence-equation names to functions; we write $\mathcal{S}[\![\mathfrak{E}]\!]$.

4.4 The extraction algorithm

We extract cost recurrences from $\mathrm{DML}_0^\Pi(\mathbb{Z})$ type derivations. A $\mathrm{DML}_0^\Pi(\mathbb{Z})$ typing judgment is of the form

$$\phi; \Gamma \vdash e : \tau,$$

where ϕ is an index context and Γ is a variable context (see Appendix A.1 for details). The central part of the extraction algorithm operates on the type derivations for the bodies of function definitions $F_1 \dots F_n$: From an expression e of first-order type that occurs within the body of a function definition, a recurrence-equation expression t of type $\mathbf{C}$ is extracted. The algorithm is defined in form of a judgment

$$\phi; \Gamma \vdash e : \rho \ \blacktriangleright\ t.$$

Consider the following definition of a function F (see Figure 8):

$$\begin{array}{l} \mathbf{let}\ F = \mathbf{fix}\ F : \Pi\, a_0 : \gamma_0 \dots \Pi\, a_l : \gamma_l\,.\, \rho_1 \to \rho_2. \\ \qquad \lambda\vec{a} : \vec{\gamma}\,.\,\mathbf{lam}\ x : \rho_1\,.\,\mathbf{case}\ x\ \mathbf{of}\ \langle x_0, \dots, x_k\rangle \Rightarrow e \end{array}$$

Assuming that e is typed as $\phi; \Gamma \vdash e : \rho_2$, the extracted recurrence equation is $F^c\ a_0 \dots a_l = t$, where t results from $\phi; \Gamma \vdash e : \rho_2 \ \blacktriangleright\ t$.

Before we give a complete description of the extraction of cost recurrences, we examine how an index context can be turned into a constraint over the declared index variables.

4.4.1 Turning an index context into a constraint

$\mathrm{DML}_0^\Pi(\mathbb{Z})$ expressions are typed under a "normal" context and an index context of form

$$\phi ::= \cdot \mid \phi, a : \gamma \mid \phi, P$$

(See Figure 3). Basically, the index context collects constraints over index variables. It is straightforward to define a function $\mathcal{C}$ that rewrites an index context ϕ into a constraint Φ such that sort definitions are "flattened out", i.e.,

$$\mathcal{C}(a : \{k : \mathbb{N} \mid k > 0\}, b : \{k : \{k' : \mathbb{N} \mid k' > 1\} \mid k \geq a\}) = a > 0 \wedge b > 1 \wedge b \geq a$$

A precise definition of $\mathcal{C}$ is deferred to Appendix B.3.2.

A useful operation over constraints is to *project out* a certain set of variables, i.e., "hide" all remaining variables by existential quantification. We write $\exists\vec{a}.\Phi$ for the constraint that results from existentially quantifying over the variables $\vec{a}$ in Φ. With a constraint solver such as used for DML type-checking, existentially quantified variables usually are simplified away.

Often equalities can be derived from a constraint Φ. Let $\vec{a}$ be a subset of the free variables in Φ; we define a substitution $\theta := \mathit{mk\_subst}_{\vec{a}}(\Phi)$ as follows: For all $a \in \vec{a}$ such that i is an index expression without free variables in $\vec{a}$, and $a = i$ can be derived from Φ, we have $\theta(a) = i$.

$$\frac{\phi;\Gamma\vdash e:\tau_1 \blacktriangleright t \quad \phi\models\tau_1\equiv\tau_2}{\phi;\Gamma\vdash e:\tau_2 \blacktriangleright t}\ \textsf{(extr-eq)}$$

$$\frac{}{\phi;\Gamma\vdash x:\tau \blacktriangleright \mathbf{0}}\ \textsf{(extr-var)} \qquad \frac{}{\phi;\Gamma\vdash \langle\rangle:\mathbf{1} \blacktriangleright \mathbf{0}}\ \textsf{(extr-unit)}$$

$$\frac{\phi;\Gamma\vdash e_1:\tau_1 \blacktriangleright t_1 \quad \phi;\Gamma\vdash e_2:\tau_2 \blacktriangleright t_2}{\phi;\Gamma\vdash \langle e_1,e_2\rangle:\tau_1\times\tau_2 \blacktriangleright t_1+t_2}\ \textsf{(extr-pair)}$$

$$\frac{\phi;\Gamma\vdash e_1:\rho_1 \blacktriangleright t_1 \quad \phi;\Gamma,x:\rho_1\vdash e_2:\rho_2 \blacktriangleright t_2}{\phi;\Gamma\vdash \mathbf{let}\ x=e_1\ \mathbf{in}\ e_2\ \mathbf{end}:\rho_2 \blacktriangleright t_1+t_2}\ \textsf{(extr-let)}$$

$$\frac{\phi;\Gamma\vdash e:\rho_1 \blacktriangleright t}{\phi;\Gamma\vdash F[i_1]\dots[i_k](e):\rho_2 \blacktriangleright t+\mathbf{c}_F+(F^c i_1\dots i_k)}\ \textsf{(extr-app)}$$

$$\frac{}{\phi;\Gamma\vdash c[i_1]\dots[i_n]:\rho \blacktriangleright \mathbf{c}_c}\ \textsf{(ty-cons-wo)}$$

$$\frac{\phi;\Gamma\vdash e:\tau \blacktriangleright t}{\phi;\Gamma\vdash c[i_1]\dots[i_n](e):\rho \blacktriangleright t+\mathbf{c}_c}\ \textsf{(ty-cons-w)}$$

$$\frac{\begin{array}{c}\phi;\Gamma\vdash e:\rho' \blacktriangleright t\\ \phi;\Gamma\vdash (p_0\Rightarrow e_0):\rho'\Rightarrow\rho \blacktriangleright br_1\\ \vdots\\ \phi;\Gamma\vdash (p_k\Rightarrow e_k):\rho'\Rightarrow\rho \blacktriangleright br_k\end{array}}{\begin{array}{l}\phi;\Gamma\vdash (\mathbf{case}\ e\ \mathbf{of}\ p_0\Rightarrow e_0\mid\dots\mid p_k\Rightarrow e_k):\rho\\ \quad\blacktriangleright t+(\mathbf{cond}\ br_1\mid br_2\mid\dots\mid br_k)\end{array}}\ \textsf{(ty-case)}$$

$$\frac{\begin{array}{l}p\downarrow\rho' \ \rhd\ (\phi';\Gamma')\\ \phi,\phi';\Gamma,\Gamma'\vdash e:\rho \blacktriangleright t\\ \Phi_1=\exists(\mathbf{dom}(\phi,\phi')\backslash\mathbf{var}(\rho')).\mathcal{C}(\phi,\phi')\\ \theta=mk\_subst_{\mathbf{dom}(\phi')}(\mathcal{C}(\phi'))\\ \Phi_2=\exists(\mathbf{dom}(\theta)).\mathcal{C}(\phi')\end{array}}{\begin{array}{l}\phi;\Gamma\vdash (p\Rightarrow e):\rho'\Rightarrow\rho\\ \quad\blacktriangleright \Phi_1\rightarrow\forall(\mathbf{dom}(\phi')\backslash\mathbf{dom}(\theta)).\Phi_2\rightarrow t[\theta]\end{array}}\ \textsf{(ty-branch)}$$

Figure 14: Extraction algorithm

4.4.2 Definition of the extraction algorithm

The definition of the judgment $\phi; \Gamma \vdash e : \rho \blacktriangleright t$ is given in Figure 14.

Most rules are fairly straightforward:

- Accessing a variable or the unit value has no cost.
- For pairs and let expressions, the total cost is the sum of the costs incurred by their subexpressions.
- The cost of executing a user-defined function F has three components: the cost of evaluating its argument, the cost $\mathbf{c}_F$ of calling the function, and the cost incurred by evaluating the function. For the latter component, a recursive call with the appropriate arguments is generated.
- The use of a constructor c costs $\mathbf{c}_c$ plus the cost of evaluating a possible argument.

A case expression is handled by converting it into a conditional. The real heart of the algorithm is the rule that handles branches of case expressions. This rule extracts a conditional branch from a type derivation for

$$\phi; \Gamma \vdash (p \Rightarrow e) : \rho' \Rightarrow \rho.$$

This judgment types the branch of a case expression that matches a pattern p against a value of type ρ'; the expression e in the branch has the same type ρ as the total case expression. We go through the premises of the corresponding extraction rule one by one:

1. The judgment $p \downarrow \rho' \;\rhd\; (\phi'; \Gamma')$ (see Figure 16 on page 34) is defined as part of the $\mathrm{DML}_0^{\Pi}(C)$ type-checking rules. It generates an index context ϕ' and a variable context Γ' that describe the index variables and normal variables occurring in the pattern p.

   For example, after translation into $\mathrm{DML}_0^{\Pi}(\mathbb{Z})$, the pattern of the second branch in `flatten` (Section 3.2) is $\mathit{lcons}[m'][n_1][n_2]\langle \mathit{xs}, \mathit{rest}\rangle$; type-checking under index context $\phi = m : \mathbb{N}, n : \mathbb{N}$ generates the following index context:

   $$\phi' = m' : \mathbb{N}, n_1 : \mathbb{N}, n_2 : \mathbb{N}, m = m' + 1, n = n_1 + n_2.$$

2. $\mathrm{DML}_0^{\Pi}(C)$ type-checking types the branch expression under the contexts ϕ, ϕ' and Γ, Γ'. From the resulting type derivation, a recurrence-equation expression t is extracted.

   For the second branch of `flatten`, this yields

   $$t = 2 + (\mathit{append}^c\ n_1\ n_2) + (\mathit{flatten}^c\ m'\ n_2)$$

3. The guard of the branch Φ_1 is derived by projecting out from $\mathcal{C}(\phi, \phi')$ over the index variables contained in ρ'.

   In `flatten`, the only index variable in $\rho' = \mathsf{llist}(m,n)$ are m and n; projecting them out yields $m > 0$. For n there is no information, because we only know that $n \in \mathbb{N}$, and (as pointed out in Section 3.1) we require all indices to be of subsorts of $\mathbb{N}$ anyways.

4. All index variables declared in ϕ' that can be expressed in terms of index variables from ϕ should be are eliminated using substitution θ: In the branch returned by the rule, the body is $t[\theta]$ rather than t.

   For the second branch of `flatten`, we can infer that $m' = m - 1$, so $\theta = [m' \mapsto m - 1]$.

5. Index variables declared in ϕ' for which no equality constraint can be derived have to be restricted. This is done by (1) hiding the variables $\mathbf{dom}(\theta)$ via existential quantification in $\mathcal{C}(\phi')$, and (2) universally quantifying over the remaining variables of $\mathbf{dom}(\phi')$, binding variables both in Φ_2 and in $t[\theta]$. Conjuncts in Φ_2 containing none of the universally quantified variables can be dropped (such conjuncts are guaranteed to be satisfied whenever Φ_1 is satisfied).

   For the second branch of `flatten`, existential quantification over m' in ϕ' yields, after normalization, $m > 0 \wedge n = n_1 + n_2$. We universally quantify over n_1 and n_2, and drop the conjunct $m > 0$.

To complete the running example: The second branch of `flatten` gives rise to the following branch of the conditional in *flatten*^{c}:

$$\begin{aligned} m > 0 \rightarrow \forall n_1, n_2 : \mathbb{N}. n_1 + n_2 = n \rightarrow \\ 2 + (\mathit{append}^c\ n_1\ n_2) + (\mathit{flatten}^c\ (m-1)\ n_2) \end{aligned}$$

4.4.3 Checking whether the bound is a recurrence

It remains to check whether we really extract a recurrence, i.e., a function defined in terms of its value on smaller arguments. For the presented language without mutual recursion (for handling mutual recursion, Xi's most recent work [14] could be adapted—see Section 5), this can be conveniently done during extraction when generating a recursive call $F^c\ i_1 \dots i_k$ inside the body that defines $F^c\ a_1 \dots a_k$: The extraction yields a recurrence if for every such call,

$$\langle i_1, \dots, i_n \rangle < \langle a_1, \dots, a_n \rangle$$

can be derived from the collected constraints for a well-founded order $<$ on tuples. In practice, one could for example fix the usual lexicographic ordering, requiring the user to enrich data correspondingly, or leave the user a choice as to which ordering should be used.

4.4.4 Correctness

Theorem 4
Let functions $F_1, \ldots, F_k$ be a well-typed block of function definitions. Let $\mathfrak{E}$ be the system of recurrence equations for $F_1^c, \ldots, F_k^c$ extracted from these definitions. Let p be a well-typed program consisting of these function definitions $F_1, \ldots, F_k$ and a program body $F_l[z_1] \ldots [z_{l'}](u)$, where $z_1, \ldots, z_{l'}$ are ground index objects and u is a value. Then:

1. *$\mathfrak{E}$ is well-typed with respect to p.*
2. *If $\mathcal{T}[\![F_l^c\ z_1 \ldots z_{l'}]\!](\mathcal{S}[\![\mathfrak{E}]\!])[] = \llcorner z' \lrcorner$ for some $z' \in \mathbf{C}$, then there exists a value v and a $z \in \mathbf{C}$ with $z \leq z'$ such that p^* evaluates to $\langle v, z \rangle$.*
3. *If the test described in Section 4.4.3 has been performed during the extraction of $\mathfrak{E}$, then the denotation of $F_l^c z_1 \ldots z_k$ under environment $\mathcal{S}[\![\mathfrak{E}]\!]$ is guaranteed to yield $\llcorner z' \lrcorner$ for some $z' \in \mathbf{C}$ rather than $\perp$.*

Part 1 and 3 of the theorem capture that the extracted system of cost bounds is well-formed: The system is well-typed, and the application of a bound F_l^c, to some legal input $z_1, \ldots z_l$ (*legal* in the sense that $F_l[z_1] \ldots [z_l](u)$ type-checks for some u) is well-defined. Because the translation $(\cdot)^*$ (see Section 4.2) captures our cost model—in the translated program, a cost counter z is calculated together with the actual result—part 2 states that this bound indeed is a cost bound for the execution of $F_l[z_1] \ldots [z_l](u)$. The proof of Theorem 4 is deferred to Appendix B.4.

5 Related work

We discuss related work regarding automated complexity analysis and type systems.

Le Métayer's ACE system [4] automatically extracts cost bounds for a subset of FP, expressing the extracted bounds as FP programs. The system is based on program transformation. The first step transforms the original program into a *step-counting* version, i.e., a program that takes the same arguments, but returns the cost of computation for these arguments rather than the result. Conceptually, this transformation corresponds to a monadic translation with the cost monad as presented in Section 4.2, where the cost component is projected out from the final result. Subsequent transformation steps try to transform the step-counting version into a composition of a cost bound and a measure function, where the measure function is composed from selector functions and the *length* function for lists. The principal goal of the ACE system is to eliminate recursion in the cost bound, which corresponds to solving recurrences; the system's library holds more than 1000 transformation rules, many of them tailored to recognize patterns of recursion. The process of finding a measure function is interleaved with the process of eliminating recursion and cannot easily be decoupled. Pointing the system to a given measure thus seems difficult. In contrast, our method separates concerns: The user can specify an appropriate measure using dependent types, but no attempts are made to solve the extracted cost recurrence.

Sands [9] treats cost analysis for higher-order call-by-name languages: Cost bounds are extracted by program transformation and reasoning over programs; his method can be seen as an extension of Le Métayer's overall approach. Sands focuses on the complications for cost analysis caused by higher-order functions and call-by-name evaluation. No special concern is given to the abstraction of data to data size, which is the main concern of our work.

Rosendahl [8] develops a system that uses abstract interpretation and program transformation techniques to extract cost bounds from a first-order subset of Scheme. An abstract interpretation is used to extend the set of S-expressions with partially known structures—unknown parts of a structure are represented by a special token that stand for all possible structures. Size measures are expressed through "inverse size functions": For a given size, an inverse size function returns a partially known structure that approximates all data structures of that size. For example, for lists, the inverse size function generates for size n a list containing n times the special token representing all possible structures. An initial cost bound for a program is achieved by (1) composing a suitable inverse size function with the abstract interpretation of the step-counting version of the program and (2) taking the term model. Program transformation is then used to simplify this initial cost bound as much as possible. Liu and Gómez [5] propose a method based on Rosendahl's work in which they use advanced program-transformation techniques to make the cost bound more efficient and more accurate. Both methods requires the user to define the abstraction from data to data size. However, with dependent types, measures can be expressed that are impossible to define with an inverse size function—the measure used in Section 3.2 for analyzing `flatten` is one example.

Reistad and Gifford [7] use an effect type system for automatically inferring cost estimates of functional programs written with combinators such as map and fold. Two indexed data-types, lists and vectors, are built into the type system. The effects associated with function types are cost expressions that may depend on indices of list/vector arguments and on cost expressions associated with function arguments. The main focus of Reistad and Gifford is to guide parallelization of programs with the inferred cost estimates.

Crary and Weirich [3] present a decidable type system for the specification and certification of resource consumption in the setting of Typed Assembly Language. The type system simulates dependent types using sum and inductive kinds. Essentially, it allows the user to annotate function arrows with resource bounds (e.g., time bounds or space bounds) in terms of the shape of data arguments and of cost bounds associated with function arguments. The focus lies on the certification of resource bounds through type checking rather than the derivation of resource bounds.

Chin and Khoo [2] propose sized types in which size information is expressed with Presburger formulas. They use a a constraint solver to infer size information. It is very likely that complexity analysis in the style of this paper could be integrated into their setting.

Recently, Xi [14] presented an extension of the DML type system that allows program termination verification. The basic observation is the same as for our work, namely that DML types can be used to encode a notion of input size. For each function, the programmer supplies a metric in terms of the input indices; if type checking succeeds, the metrics are guaranteed to specify a termination order. Both higher-order functions and general recursion are handled. We believe that our work would benefit substantially from reformulating it in this extended type system: As an immediate benefit, extracted cost bounds could be easily verified to be recurrences also for general recursion. Also an extension of this work to higher-order functions, if possible, should be easier within Xi's new type system, though of course our work handles defunctionalized programs (à la Reynolds) and thus, indirectly, higher-order functions.

6 Conclusion

We have presented a method for automatically extracting cost recurrences from first-order DML programs. The distinct feature of our method is the use of dependent types to describe a size measure that abstracts from data to data size. The user has to choose an appropriate size measure for our method to successfully extract a cost recurrence. Because of the high expressiveness of its types, DML offers high flexibility for tailoring size measures. The required DML type annotations usually are easy to find: Because size measures encode shape information of data types, they closely correspond to the programmer's intuitive understanding of how his program works. Our method harnesses this intuition for automatic cost analysis.

Acknowledgments I am indebted to Olivier Danvy, Julia Lawall and Zhe Yang for their encouragement and fruitful discussions on the subject of this work. Further thanks are due to Olivier Danvy, Andrzej Filinski and Julia Lawall for their numerous constructive comments. I am also grateful to the anonymous referees and Neil Jones for further comments.

References

[1] Lennart Augustsson. Cayenne—a language with dependent types. In Paul Hudak and Christian Queinnec, editors, *Proceedings of ICFP'98*, pages 239–250, Baltimore, Maryland, September 1998. ACM Press.

[2] Wei-Ngan Chin and Siau-Cheng Khoo. Calculating sized types. *Higher-Order and Symbolic Computation*, 14(2/3), 2001. To appear.

[3] Karl Crary and Stephanie Weirich. Resource bound certification. In Thomas Reps, editor, *Proceedings of POPL'00*, pages 184–198, Boston Massachusetts, January 2000. ACM Press.

[4] Daniel Le Métayer. ACE: An automatic complexity evaluator. *ACM Transactions on Programming Languages and Systems*, 10(2):248–266, April 1988.

[5] Yanhong Annie Liu and Gustavo Gómez. Automatic accurate time-bound analysis for high-level languages. In Frank Mueller and Azer Bestavros, editors, *Proceedings of the ACM SIGPLAN 1998 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, number 1474 in LNCS, pages 31–40, Montréal, Canada, June 1998. Springer-Verlag.

[6] Eugenio Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[7] Brian Reistad and David K. Gifford. Static dependent costs for estimating program execution time. In Carolyn L. Talcott, editor, *Proceedings of LFP'94*, LISP Pointers, Vol. VII, No. 3, Orlando, Florida, June 94. ACM Press.

[8] Mads Rosendahl. Automatic complexity analysis. In Joseph E. Stoy, editor, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture '89*, pages 144–156, London, September 1989. ACM Press.

[9] David Sands. *Calculi for Time Analysis of Functional Programs*. PhD thesis, Department of Computing, Imperial College, University of London, September 1990.

[10] Philip Wadler. The essence of functional programming. In Andrew W. Appel, editor, *Proceedings of POPL'92*, pages 1–14, Albuquerque, New Mexico, January 1992. ACM Press.

[11] Hongwei Xi. de Caml. A prototype implementation of DML, based on Caml-light. Available from `http://www.ececs.uc.edu/~hwxi/DML/DML.html`.

[12] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998.

[13] Hongwei Xi. Dependently typed data structures. In Chris Okasaki, editor, *Proceedings of Workshop of Algorithmic Aspects of Advanced Programming Languages (WAAAPL '99)*, pages 17–32, Paris, September 1999. Available from `http://www.cs.columbia.edu/~cdo/waaapl99.pdf`.

[14] Hongwei Xi. Dependent types for program termination verification. In *Proceedings of 16th IEEE Symposium on Logic in Computer Science*, Boston, Massachusetts, June 2001. IEEE Computer Society Press.

[15] Hongwei Xi and Frank Pfenning. Eliminating array bound checking through dependent types. In Keith D. Cooper, editor, *Proceedings of PLDI'98*, pages 249–257, Montreal, June 1998. ACM Press.

[16] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In Alex Aiken, editor, *Proceedings of POPL'99*, pages 214–227, San Antonio, Texas, January 1999. ACM Press.

A DML

In the following we give a short overview over the formalization of $\mathrm{DML}_0^\Pi(C)$ that is used in this article. We gloss over details such as type-formation rules, well-formedness of contexts, and type-equivalence. The complete formalization can be found in Xi's PhD thesis [12, Chapters 2–4].

A.1 DML typing rules

A typing judgment for $\mathrm{DML}_0^\Pi(C)$ has the form

$$\phi; \Gamma \vdash e : \tau,$$

where ϕ is an index context and Γ a (normal) context; typing is with respect to a signature $\mathcal{S}$ that assigns types to constructors. In the following, we focus on typing rules that treat indices—the remaining rules are fairly standard.

The treatment of indices is based on a judgment $\phi \vdash i : \gamma$ that expresses that $i : \gamma$ can be derived from the index context ϕ; to establish this judgment, constraint solving is required. The judgment is used, for example, to express type-soundness of an index substitution θ under a context ϕ as $\phi \vdash \theta : \phi'$ (Figure 15a), which basically says that, assuming the context given by ϕ substitution θ assigns to all index variables declared in ϕ' an index object of the declared sort. The following lemma [12, Chapter 3] shows a useful link between type-soundness with respect to an index context ϕ and satisfyability with respect to ϕ:

Lemma 5
Let ϕ and ϕ' be index contexts and θ an index substitution such that $\phi \vdash \theta : \phi'$ is derivable. Then, if $\phi, \phi' \models \Phi$ is derivable, also $\phi \models \Phi[\theta]$ is derivable.

Figure 15b defines a corresponding judgment for general substitutions; it is straightforward to show that if $\phi; \Gamma \vdash \theta : (\phi'; \Gamma')$ holds for a substitution θ, then $\phi \vdash \theta_\phi : \phi'$ holds for its restriction θ_ϕ to index variables.

Figure 16 and Figure 17 on page 36 display the typing rules for $\mathrm{DML}_0^\Pi(C)$. The judgment $p \downarrow \tau \ \rhd \ (\phi; \Gamma)$ defined in Figure 16 on the following page is used for typing pattern matching over an expression of type τ: Type information about variables and index variables occurring in pattern p is gathered. Figure 17 on page 36 defines the "top-level" typing judgment $\phi; \Gamma \vdash e : \tau$ for $\mathrm{DML}_0^\Pi(C)$. The rule for type equality ty-eq uses a judgment $\phi \models \tau_1 \equiv \tau_2$ that is defined as

$$\frac{}{\phi \vdash [\,] : \cdot}\ \textsf{(isubst-empty)}$$

$$\frac{\phi \vdash \theta : \phi' \quad \phi \vdash i : \gamma[\theta]}{\phi \vdash \theta[a \mapsto i] : \phi', a : \gamma}\ \textsf{(isubst-ivar)}$$

$$\frac{\phi \vdash \theta : \phi' \quad \phi \models P[\theta]}{\phi \vdash \theta : \phi', P}\ \textsf{(subst-iprop)}$$

a: Type-soundness of an index substitution

$$\frac{}{\phi; \Gamma \vdash [\,] : (\cdot; \cdot)}\ \textsf{(subst-empty)}$$

$$\frac{\phi; \Gamma \vdash \theta : (\phi'; \Gamma') \quad \phi; \Gamma \vdash e : \tau[\theta]}{\phi; \Gamma \vdash \theta[x \mapsto e] : (\phi'; \Gamma', x : \tau)}\ \textsf{(subst-var)}$$

$$\frac{\phi; \Gamma \vdash \theta : (\phi'; \Gamma') \quad \phi \vdash i : \gamma[\theta]}{\phi; \Gamma \vdash \theta[a \mapsto i] : (\phi', a : \gamma; \Gamma')}\ \textsf{(subst-ivar)}$$

$$\frac{\phi; \Gamma \vdash \theta : (\phi'; \Gamma') \quad \phi \models P[\theta]}{\phi; \Gamma \vdash \theta : (\phi', P; \Gamma')}\ \textsf{(subst-iprop)}$$

b: Type-soundness of a substitution (general)

Figure 15: Type-soundness of a substitution

$$\frac{}{x \downarrow \tau \ \rhd\ (\cdot; x : \tau)}\ \textsf{(pat-var)} \qquad \frac{}{\langle\rangle \downarrow \mathbf{1} \ \rhd\ (\cdot; \cdot)}\ \textsf{(pat-unit)}$$

$$\frac{p_1 \downarrow \tau_1 \ \rhd\ (\phi_1; \Gamma_1) \quad p_2 \downarrow \tau_2 \ \rhd\ (\phi_2; \Gamma_2)}{\langle p_1, p_2 \rangle \downarrow \tau_1 \times \tau_2 \ \rhd\ (\phi_1, \phi_2; \Gamma_1, \Gamma_2)}\ \textsf{(pat-prod)}$$

$$\frac{\mathcal{S}(c) = \Pi\, a_1 : \gamma_1 \ldots \Pi\, a_k : \gamma_k \,.\, \delta(i)}{c[a_1] \ldots [a_k] \downarrow \delta(j) \ \rhd\ (a_1 : \gamma_1, \ldots, a_k : \gamma_k, i = j; \cdot)}\ \textsf{(pat-cons-wo)}$$

$$\frac{\mathcal{S}(c) = \Pi\, a_1 : \gamma_1 \ldots \Pi\, a_k : \gamma_k \,.\, \tau \to \delta(i) \quad p \downarrow \tau \ \rhd\ (\phi; \Gamma)}{c[a_1] \ldots [a_k](p) \downarrow \delta(j) \ \rhd\ (a_1 : \gamma_1, \ldots, a_k : \gamma_k, i = j, \phi; \Gamma)}\ \textsf{(pat-cons-w)}$$

Figure 16: Typing rules for patterns in $\mathrm{DML}_0^{\Pi}(C)$

the congruent extension of $\phi \models i = j$ from index objects to types. The rules ty-cons-wo, ty-cons-w and ty-iapp, for constructors and application to an index object, respectively, examine whether index objects are indeed of the required sort; when typing a constructor, the judgment for establishing type soundness of index substitutions is used to account for possible sequential dependencies among the sorts pertaining to the arguments of the constructor.

The following theorem [12, Chapter 4] shows that types are preserved under a type-sound substitution.

Theorem 6 (Substitution)
If $\phi, \phi'; \Gamma, \Gamma' \vdash e : \tau$ and $\phi; \Gamma \vdash \theta : (\phi'; \Gamma')$ are derivable, then $\phi; \Gamma \vdash e[\theta] : \tau[\theta]$ is derivable.

A.2 DML semantics

Figure 18 on page 37 describes a natural semantics for $\mathrm{DML}_0^\Pi(C)$: $e \longrightarrow v$ means that e reduces to a value v, where

$$v ::= c[i_1] \ldots [i_k] \mid c[i_1] \ldots [i_k](v) \mid \langle\rangle \mid \langle v_1, v_2 \rangle \mid (\mathbf{lam}\ x : \tau \,.\, e) \mid (\lambda a : \gamma \,.\, v).$$

Notice that type indices are never evaluated. The language design decision is that there is no direct interaction between indices and code execution; type indices are used only for type-checking.

The rule that describes the semantics of a case expression makes use of a judgment $\mathbf{match}(v, p) \Longrightarrow \theta$ defined in Figure 19 on page 38: If a value v matches a pattern p, a substitution for the free variables in p is returned. Notice that the semantics of case expressions is nondeterministic: an arbitrary matching arm is picked.

Xi [12, Chapter 4] proves the following theorem connecting the type system and the semantics:

Theorem 7 (Relating types and semantics)

1. *Assume that there is no $a \in \mathbf{dom}(\phi)$ that occurs in pattern p. If $\phi; \Gamma \vdash v : \tau$, $p \downarrow \tau \ \rhd\ (\phi'; \Gamma')$ and $\mathbf{match}(p, v) \Longrightarrow \theta$, then $\phi; \Gamma \vdash \theta : (\phi'; \Gamma')$ is derivable.*

2. *Given e,v in $\mathrm{DML}_0^\Pi(C)$ such that $e \longrightarrow v$ is derivable. If $\phi; \Gamma \vdash e : \tau$ is derivable, then $\phi; \Gamma \vdash v : \tau$ is derivable.*

$$\frac{\phi;\Gamma \vdash e : \tau_1 \quad \phi \models \tau_1 \equiv \tau_2}{\phi;\Gamma \vdash e : \tau_2}\ \textsf{(ty-eq)} \qquad \frac{\Gamma(x) = \tau}{\phi;\Gamma \vdash x : \tau}\ \textsf{(ty-var)}$$

$$\frac{\begin{array}{c}\mathcal{S}(c) = \Pi\, a_1 : \gamma_1 \ldots \Pi\, a_k : \gamma_k \,.\, \delta(i) \\ \phi \vdash [a_1, \ldots, a_k \mapsto i_1, \ldots, i_k] : (a_1 : \gamma_1, \ldots, a_k : \gamma_k)\end{array}}{\phi;\Gamma \vdash c[i_1] \ldots [i_k] : \delta(i[a_1, \ldots, a_k \mapsto i_1, \ldots, i_k])}\ \textsf{(ty-cons-wo)}$$

$$\frac{\begin{array}{c}\mathcal{S}(c) = \Pi\, a_1 : \gamma_1 \ldots \Pi\, a_k : \gamma_k \,.\, \tau \to \delta(i) \\ \phi \vdash [a_1, \ldots, a_k \mapsto i_1, \ldots, i_k] : (a_1 : \gamma_1, \ldots, a_k : \gamma_k) \\ \phi;\Gamma \vdash e : \tau[a_1, \ldots, a_k \mapsto i_1, \ldots, i_k]\end{array}}{\phi;\Gamma \vdash c[i_1] \ldots [i_k](e) : \delta(i[a_1, \ldots, a_k \mapsto i_1, \ldots, i_k])}\ \textsf{(ty-cons-w)}$$

$$\frac{}{\phi;\Gamma \vdash \langle\rangle : \mathbf{1}}\ \textsf{(ty-unit)} \qquad \frac{\phi;\Gamma \vdash e_1 : \tau_1 \quad \phi;\Gamma \vdash e_2 : \tau_2}{\phi;\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}\ \textsf{(ty-prod)}$$

$$\frac{\begin{array}{c}\phi;\Gamma \vdash e : \tau' \\ \phi;\Gamma \vdash (p_1 \Rightarrow e_1) : \tau' \Rightarrow \tau \\ \vdots \\ \phi;\Gamma \vdash (p_k \Rightarrow e_k) : \tau' \Rightarrow \tau\end{array}}{\phi;\Gamma \vdash (\mathbf{case}\ e\ \mathbf{of}\ p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k) : \tau}\ \textsf{(ty-case)}$$

$$\frac{p \downarrow \tau' \ \rhd\ (\phi';\Gamma') \qquad \phi, \phi';\Gamma, \Gamma' \vdash e : \tau}{\phi;\Gamma \vdash (p \Rightarrow e) : \tau' \Rightarrow \tau}\ \textsf{(ty-branch)}$$

$$\frac{\phi, a : \gamma;\Gamma \vdash e : \tau}{\phi;\Gamma \vdash (\lambda a : \gamma \,.\, e) : \Pi\, a : \gamma \,.\, \tau}\ \textsf{(ty-ilam)}$$

$$\frac{\phi;\Gamma \vdash e : \Pi\, a : \gamma \,.\, \tau \qquad \phi \vdash i : \gamma}{\phi;\Gamma \vdash e[i] : \tau[a \mapsto i]}\ \textsf{(ty-iapp)}$$

$$\frac{\phi;\Gamma, x : \tau_1 \vdash e : \tau_2}{\phi;\Gamma \vdash (\mathbf{lam}\ x : \tau_1 \,.\, e) : \tau_1 \to \tau_2}\ \textsf{(ty-lam)}$$

$$\frac{\phi;\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \phi;\Gamma \vdash e_2 : \tau_1}{\phi;\Gamma \vdash e_1(e_2) : \tau_2}\ \textsf{(ty-app)}$$

$$\frac{\phi;\Gamma \vdash e_1 : \tau_1 \quad \phi;\Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\phi;\Gamma \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} : \tau_2}\ \textsf{(ty-let)}$$

$$\frac{\phi;\Gamma, f : \tau \vdash e : \tau}{\phi;\Gamma \vdash (\mathbf{fix}\ f : \tau.e) : \tau}\ \textsf{(ty-fix)}$$

Figure 17: Typing rules for $\mathrm{DML}_0^{\Pi}(C)$

$$\frac{}{c[i_1]\ldots[i_k] \longrightarrow c[i_1]\ldots[i_k]}\ \textsf{(ev-cons-wo)}$$

$$\frac{e \longrightarrow v}{c[i_1]\ldots[i_k](e) \longrightarrow c[i_1]\ldots[i_k](v)}\ \textsf{(ev-cons-w)}$$

$$\frac{e_1 \longrightarrow v_1 \quad e_2 \longrightarrow v_2}{\langle e_1, e_2\rangle \longrightarrow \langle v_1, v_2\rangle}\ \textsf{(ev-prod)}$$

$$\frac{e_0 \longrightarrow v_0 \quad \mathbf{match}(v_0, p_l) \Longrightarrow \theta \text{ for some } 1 \leq l \leq k \quad e_l[\theta] \longrightarrow v}{\mathbf{case}\ e_0\ \mathbf{of}\ (p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k) \longrightarrow v}\ \textsf{(ev-case)}$$

$$\frac{e \longrightarrow v}{(\lambda a : \gamma\,.\,e) \longrightarrow (\lambda a : \gamma\,.\,v)}\ \textsf{(ev-ilam)}$$

$$\frac{e \longrightarrow (\lambda a : \gamma\,.\,v)}{e[i] \longrightarrow v[a \mapsto i]}\ \textsf{(ev-iapp)}$$

$$\frac{}{(\mathbf{lam}\ x : \tau\,.\,e) \longrightarrow (\mathbf{lam}\ x : \tau\,.\,e)}\ \textsf{(ev-lam)}$$

$$\frac{e_1 \longrightarrow (\lambda x : \tau\,.\,e) \quad e_2 \longrightarrow v_2 \quad e[x \mapsto v_2] \longrightarrow v}{e_1(e_2) \longrightarrow v}\ \textsf{(ev-app)}$$

$$\frac{e_1 \longrightarrow v_1 \quad e_2[x \mapsto v_1] \longrightarrow v}{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} \longrightarrow v}\ \textsf{(ev-let)}$$

$$\frac{}{(\mathbf{fix}\ f : \tau.e) \longrightarrow e[f \mapsto (\mathbf{fix}\ f : \tau.e)]}\ \textsf{(ev-fix)}$$

Figure 18: Natural Semantics of $\mathrm{DML}_0^{\Pi}(C)$

$$\frac{}{\mathbf{match}(x, v) \Longrightarrow [x \mapsto v]} \ (\textsf{mat-var})$$

$$\frac{}{\mathbf{match}(\langle\rangle, \langle\rangle) \Longrightarrow []} \ (\textsf{mat-unit})$$

$$\frac{\mathbf{match}(p_1, v_1) \Longrightarrow \theta_1 \quad \mathbf{match}(p_2, v_2) \Longrightarrow \theta_2}{\mathbf{match}(\langle p_1, p_2\rangle, \langle v_1, v_2\rangle) \Longrightarrow \theta_1\theta_2} \ (\textsf{mat-prod})$$

$$\frac{}{\mathbf{match}(c[a_1] \ldots [a_k], c[i_1] \ldots [i_k]) \Longrightarrow [a_1 \mapsto i_1, \ldots, a_k \mapsto i_k]} \ (\textsf{mat-cons-wo})$$

$$\frac{\mathbf{match}(p, v) \Longrightarrow \theta}{\mathbf{match}(c[a_1] \ldots [a_k](p), c[i_1] \ldots [i_k](v)) \Longrightarrow \theta[a_1 \mapsto i_1, \ldots, a_k \mapsto i_k]} \ (\textsf{mat-cons-w})$$

Figure 19: Semantics of pattern matching in $\mathrm{DML}_0^{\Pi}(C)$

B Formal development

B.1 A modified semantics a first-order fragment of DML

Figure 20 displays a modified semantics in which an environment of function definitions is maintained. Rule ev-mod-fundef shows how the environment is built up from function definitions, rule ev-mod-fapp shows how the environment is used. The judgment $\mathbf{match}(v, p) \Longrightarrow \theta$ in rule ev-mod-case is defined as in the standard semantics (see Figure 19 on the page before): If value v can be matched against pattern p, the corresponding substitution θ of the free variables in p is returned.

We present a proof of Theorem 1 on page 18, which states that the origi-

$$\frac{}{c[i_1]\ldots[i_k] \longrightarrow_\Theta c[i_1]\ldots[i_k]}\ \textsf{(ev-mod-cons-wo)}$$

$$\frac{e \longrightarrow_\Theta v}{c[i_1]\ldots[i_k](e) \longrightarrow_\Theta c[i_1]\ldots[i_k](v)}\ \textsf{(ev-mod-cons-w)}$$

$$\frac{e_1 \longrightarrow_\Theta v_1 \quad e_2 \longrightarrow_\Theta v_2}{\langle e_1, e_2\rangle \longrightarrow_\Theta \langle v_1, v_2\rangle}\ \textsf{(ev-mod-prod)}$$

$$\frac{\begin{array}{l} e_0 \longrightarrow_\Theta v_0 \\ \mathbf{match}(v_0, p_l) \Longrightarrow \theta \text{ for some } 1 \leq l \leq k \\ e_l[\theta] \longrightarrow_\Theta v \end{array}}{\mathbf{case}\ e_0\ \mathbf{of}\ (p_1 \Rightarrow e_1 \mid \ldots \mid p_k \Rightarrow e_k) \longrightarrow_\Theta v}\ \textsf{(ev-mod-case)}$$

$$\frac{e_1 \longrightarrow_\Theta v_1 \quad e_2[x \mapsto v_1] \longrightarrow_\Theta v}{\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2\ \mathbf{end} \longrightarrow_\Theta v}\ \textsf{(ev-mod-let)}$$

$$\frac{e \longrightarrow_{\Theta[F \mapsto e_F]} v}{\mathbf{let}\ F = \mathbf{fix}\ F : \tau . e_F\ \mathbf{in}\ e\ \mathbf{end} \longrightarrow_\Theta v}\ \textsf{(ev-mod-fundef)}$$

$$\frac{\begin{array}{l} e \longrightarrow_\Theta v' \\ \Theta(F) = \lambda\vec{a} : \vec{\gamma}\,.\,\mathbf{lam}\ x : \rho\,.\,\mathit{body} \\ \mathit{body}[\vec{a} \mapsto \vec{\imath}][x \mapsto v'] \longrightarrow_\Theta v \end{array}}{F[\vec{\imath}](e) \longrightarrow_\Theta v}\ \textsf{(ev-mod-fapp)}$$

Figure 20: Natural Semantics of first-order fragment of DML

nal semantics of $\mathrm{DML}_0^\Pi(C)$ from Appendix A.2 and the modified semantics of Figure 20 on the page before are equivalent.

Given a program p of the form described in Figure 8 on page 19, the modified semantics builds up an environment from the function definitions in p (rule ev-mod-fundef), while the original semantics accumulates a substitution (rule ev-let). The judgment $\vdash \Theta : \Gamma$ (Figure 21) is used to establish the well-formedness of an environment Θ and to type the functions contained in it.

$$\frac{}{\vdash [] : \cdot}\ (\text{ty-env-nil})$$

$$\frac{\vdash \Theta : \Gamma \quad \cdot; \Gamma, F : \tau \vdash \lambda\vec{a} : \vec{\gamma} \,.\, \mathbf{lam}\ x : \rho_F \,.\, \mathit{body}_F : \tau}{\vdash \Theta[F \mapsto \lambda\vec{a} : \vec{\gamma} \,.\, \mathbf{lam}\ x : \rho_F \,.\, \mathit{body}_F] : \Gamma, F : \tau}\ (\text{ty-env-cons})$$

Figure 21: Typing an environment

We further use a mapping $(\cdot)^\circ$ to relate the environment of the modified semantics with the substitution of the original semantics. Given an environment $\vdash \Theta : \Gamma$, the substitution Θ° is defined as follows:

$$\begin{array}{rcl} ([])^\circ & = & [] \\ (\Theta[F \mapsto e])^\circ & = & [F \mapsto \mathbf{fix}\ F : \Gamma(F).e] \circ \Theta^\circ \end{array}$$

The following lemma relates the modified semantics with the original semantics.

Lemma 8
Let $\vdash \Theta : \Gamma_F$ and let e be a body expression such that $\phi; \Gamma_F, \Gamma \vdash e : \rho$ is derivable. Then $e[\Theta^\circ] \longrightarrow v$ iff $e \longrightarrow_\Theta v$.

Proof: The proof of the lemma is conducted by structural induction over the derivation $e[\Theta^\circ] \longrightarrow v$ (proving the implication from left to right) and $e \longrightarrow_\Theta v$ (right to left).

We show the implication from left to right, examining the last rule in a derivation of $e[\Theta^\circ] \longrightarrow v$. For rules ev-cons-wo, ev-cons-w, ev-prod, ev-case and ev-let, the lemma follows immediately by induction hypothesis. For rules ev-ilam, ev-iapp, ev-app and ev-fix observe, that because of the restricted shape of e (see Figure 8 on page 19), only rule ev-ilam can occur as the last rule in a derivation $e[\Theta^\circ] \longrightarrow v$, namely for $e = F[\vec{\imath}](e')$. Assume therefore that $(F[\vec{\imath}](e'))[\Theta^\circ] \longrightarrow v$ is derivable; an analysis of the shape of the corresponding derivation (see Figure 18 on page 37 for the rules) shows that for some value v_1 there are derivations of $e'[\Theta^\circ] \longrightarrow v_1$ and $\mathit{body}[\vec{a} \mapsto \vec{\imath}][x \mapsto v_1][\Theta^\circ] \longrightarrow v$. Using the induction hypothesis, we can derive that $\mathit{body}[\vec{a} \mapsto \vec{\imath}][x \mapsto v_1] \longrightarrow_\Theta v$, from which it follows easily with rule ev-mod-fapp (Figure 20 on the preceding page) that $F[\vec{\imath}](e') \longrightarrow_\Theta v$.

The other direction of the implication follows similarly. □

We are now in a position to prove Theorem 1 on page 18.

Proof: Let program p be of the form given in Figure 8 on page 19 with function definitions $F_l = \textbf{fix}\ F_l : \tau_l.e_l$ for $0 \leq l \leq k$ and program body e. With the typing rules $\mathsf{ty\text{-}let}$ and $\mathsf{ty\text{-}fix}$ (Figure 17 on page 36) and the rules from Figure 21 on the preceding page a straightforward induction over the number of function definitions shows that

$$\vdash (F_1 : \tau_1, \ldots, F_k : \tau_k) : \overbrace{[F_1 \mapsto e_1, \ldots, F_k \mapsto e_k]}^{=:\Theta}.$$

Further, examining the modified and original semantics, we see:

$$\begin{aligned} \mathsf{p} \longrightarrow v \quad &\text{iff} \quad e[\Theta^\circ] \longrightarrow v \\ \mathsf{p} \longrightarrow_{[]} v \quad &\text{iff} \quad e \longrightarrow_\Theta v \end{aligned}$$

With Lemma 8 on the page before it follows that $\mathsf{p} \longrightarrow v$ iff $\mathsf{p} \longrightarrow_{[]} v$. □

B.2 The monadic translation

We present a proof of Theorem 2 on page 20, which states that the monadic translation preserves types and semantics.

B.2.1 The monadic translation preserves types

The following Lemma expresses the validity of the typing rules given for $\textbf{val}^{\mathsf{C}}$, $\textbf{let}^{\mathsf{C}}$ and $\textbf{cost}$ in Section 4.2 with respect to the expansion of these constructs as given in Figure 10 on page 21.

Lemma 9
The typing rules for $\textbf{val}^{\mathsf{C}}$, $\textbf{let}^{\mathsf{C}}$ *and* $\textbf{cost}$ *as given in Section 4.2 are admissible (assuming that* $\textbf{val}^{\mathsf{C}}$, $\textbf{let}^{\mathsf{C}}$ *and* $\textbf{cost}$ *are expanded as described in Figure 10 on page 21).*

Proof: Straightforward by constructing the corresponding type derivations. □

The first part of Theorem 2 on page 20 follows immediately from the following lemma.

Lemma 10
For an expression e, if $\phi; \Gamma \vdash e : \rho$ is derivable, then $\phi; \Gamma^* \vdash e^* : \mathsf{C}\,\rho$ is derivable, where Γ^* results from Γ by wrapping the result type of every function declared in Γ with C.

Proof: The proof is conducted by structural induction on the derivation of $\phi; \Gamma \vdash e : \rho$ (typing rules in Figure 17 on page 36). If the last rule of the derivation is $(\textsf{ty-eq})$, then the lemma follows immediately by induction hypothesis. Otherwise, because the remaining rules are syntax directed, we proceed by examining all possible expressions e.

We present one interesting case, namely a function call $\phi; \Gamma \vdash F[\vec{\imath}](e) : \rho$. Examining the possible derivations, we see that there exists ρ_1 such that $\phi; \Gamma \vdash F\ [\vec{\imath}] : \rho_1 \to \rho$ and $\phi; \Gamma \vdash e : \rho_1$ are derivable. The monadic translation of $F[\vec{\imath}](e)$ is

$$\begin{array}{l}\mathbf{let}^{\mathsf{C}}\ x = e^* \\ \mathbf{in\ cost}_{\mathbf{c}_c}\ (F[\vec{\imath}](x))\ \mathbf{end}\end{array}$$

Because of Lemma 9 on the page before, rule ty-monadic-let (Section 4.2) is admissible, hence we need to find derivations of $\phi; \Gamma \vdash e^* : \mathsf{C}\,\rho_1$ and $\phi; \Gamma^*, x : \rho_1 \vdash \mathbf{cost}_{c_F}\ (F[\vec{\imath}](x)) : \mathsf{C}\,\rho$. The former follows by induction hypothesis, the latter is easily derived using rule ty-monadic-cost and the fact that $\phi; \Gamma \vdash F\ [\vec{\imath}] : \rho_1 \to \rho$ is derivable. □

B.2.2 The monadic translation preserves semantics

In order to prove the second part of Theorem 2 on page 20, namely that the monadic translation preserves semantics, we need the following lemma:

Lemma 11
Let $\vdash \Theta : \Gamma_F$ and $\phi; \Gamma_F, \Gamma \vdash e : \rho$ be derivable. Then $e \longrightarrow_\Theta v$ iff there exists a $z \in \mathbf{C}$ such that $e^* \longrightarrow_{\Theta^*} \langle v, z\rangle$ is derivable (where $\Theta^*(F) = \Theta(F)^*$ for all $F \in \mathbf{dom}(\Theta)$).

Proof: The proof is conducted by structural induction over $e \longrightarrow_\Theta v$ (proving the implication from left to right) and over $e^* \longrightarrow_{\Theta^*} \langle v, z\rangle$ (right to left).

We show the case of a function call $F[\vec{\imath}](e)$. Assume that there exists a derivation of $F[\vec{\imath}](e) \longrightarrow_\Theta v$. Rule inversion with ev-mod-fapp (Figure 20 on page 39) shows that there exists a value v' such that $e \longrightarrow_\Theta v'$ and $\mathit{body}[\vec{a} \mapsto \vec{\imath}][x_1 \mapsto v'] \longrightarrow_\Theta v$ are derivable, where $\Theta(F) = \lambda\vec{a} : \vec{\gamma} \,.\, \mathbf{lam}\ x : \rho \,.\, \mathit{body}$. Consider now the translated term $(F[\vec{\imath}](e))^*$ where the monadic constructs have been expanded as shown in Figure 10 on page 21:

$$\begin{array}{l}\mathbf{case}\ e^*\ \mathbf{of} \\ \quad \langle x_1, z_1\rangle \Rightarrow\ \mathbf{case}\ (\mathbf{case}\ F[\vec{\imath}](x_1)\ \mathbf{of}\ \langle x_3, z_3\rangle \Rightarrow \langle x_3, z_3 + c_F\rangle)\ \mathbf{of} \\ \qquad\qquad\qquad \langle x_2, z_2\rangle \Rightarrow\ \langle x_2, z_1 + z_2\rangle\end{array}$$

By induction hypothesis, we know that there is a z'_1 such that $e^* \longrightarrow_{\Theta^*} \langle v', z'_1\rangle$; with ev-mod-case it follows that we need to show a derivation of $F[\vec{\imath}](v') \longrightarrow_{\Theta^*} \langle v, z'_3\rangle$ for some $z'_3 \in \mathbf{C}$. Such a derivation can be constructed using ev-mod-fapp and a derivation $\mathit{body}^*[\vec{a} \mapsto \vec{\imath}][x_1 \mapsto v'] \longrightarrow_{\Theta^*} \langle v, z'_3\rangle$, which exists by induction hypothesis. □

We are now in a position to prove the second part of Theorem 2 on page 20:

Proof: Let program p be of the form given in Figure 8 on page 19 with function definitions $F_l = \mathbf{fix}\ F_l : \tau_l.e_l$ for $0 \leq l \leq k$ and program body e. Let

$$\begin{array}{rcl} \Gamma & := & F_1 : \tau_1, \ldots, F_k : \tau_k \\ \Theta & := & [F_1 \mapsto e_1, \ldots, F_k \mapsto e_k] \end{array}$$

Inspecting the type derivation of p it is easy to see that $\vdash \Theta : \Gamma$ and $\cdot; \Gamma \vdash e : \rho$ are derivable. From the semantics we know that $p \longrightarrow_{[]} v$ and $p^* \longrightarrow_{[]} \langle v', z\rangle$ iff $e \longrightarrow_{\Theta} v$ and $e^* \longrightarrow_{\Theta^*} \langle v', z\rangle$, respectively. With Lemma 11 on the page before we have that $e \longrightarrow_{\Theta} v$ iff $e^* \longrightarrow_{\Theta^*} \langle v, z\rangle$, which concludes the proof. □

B.3 Extraction of recurrence equations—preliminaries

B.3.1 An erasure from index sorts to index types

The erasure $\widetilde{\cdot}$ maps an index sort to its associated index type by removing all constraint-related information. It is defined as follows:

$$\begin{array}{rcl} \widetilde{\mathbb{N}} & = & \mathbb{N} \\ \widetilde{\mathbf{1}} & = & \mathbf{1} \\ \widetilde{\gamma_1 \times \gamma_2} & = & \widetilde{\gamma_1} \times \widetilde{\gamma_2} \\ \widetilde{\{a : \gamma \mid P\}} & = & \widetilde{\gamma} \end{array}$$

For a context Γ that maps function names to types we define $\widetilde{\Gamma}$ such that

$$\widetilde{\Gamma}(F^c) = \widetilde{\gamma_0} \rightarrow \widetilde{\gamma_1} \rightarrow \ldots \widetilde{\gamma_k} \rightarrow \mathbf{C}$$

if

$$\Gamma(F) = \Pi\, a_0 : \gamma_0 \,.\, \Pi\, a_1 : \gamma_1 \ldots \Pi\, a_k : \gamma_k \,.\, \rho_1 \rightarrow \rho_2.$$

For an index context ϕ we define $\widetilde{\phi}$ as follows:

$$\begin{array}{rcl} \widetilde{\cdot} & = & \cdot \\ \widetilde{\phi, a : \gamma} & = & \widetilde{\phi}, a : \widetilde{\gamma} \\ \widetilde{\phi, P} & = & \widetilde{\phi} \end{array}$$

The following lemma shows that the erasure preserves type soundness of index substitutions.

Lemma 12
If $\phi \vdash \theta : \phi'$ is derivable, then $\widetilde{\phi} \vdash \theta : \widetilde{\phi'}$ is derivable.

Proof: With a straightforward induction over the structure of sort γ, it is easy to show that if $\phi \vdash i : \gamma$ is derivable, then $\widetilde{\phi} \vdash i : \widetilde{\gamma}$ is derivable. Using this fact, one then can show the lemma with a straightforward induction over the length of ϕ'. □

B.3.2 Flattening index contexts into constraints

The algorithm for extracting cost recurrences uses a function $\mathcal{C}$ to rewrite an index context into a conjunctive constraint by flattening subsort definitions. A declaration $i : \{k : \gamma \mid P\}$ is rewritten with the conjunction of (1) the constraints imposed by the declaration $i : \gamma$ and (2) the index proposition $P[k \mapsto i]$. The resulting constraint $\Phi := \mathcal{C}(\phi)$ is a quantifier-free conjunction of equality constraints and index propositions:[7]

$$\begin{array}{lcl}
\mathcal{C}(\cdot) & = & \top \\
\mathcal{C}(\phi, i : \mathbf{1}) & = & \mathcal{C}(\phi) \\
\mathcal{C}(\phi, i : \mathbb{N}) & = & \mathcal{C}(\phi) \\
\mathcal{C}(\phi, i : \gamma_1 \times \gamma_2) & = & \mathcal{C}(\phi, \mathbf{fst}(i) : \gamma_1, \mathbf{snd}(i) : \gamma_2) \\
\mathcal{C}(\phi, i : \{k : \gamma \mid P\}) & = & \mathcal{C}(\phi, i : \gamma) \wedge P[k \mapsto i] \\
\mathcal{C}(\phi, P) & = & \mathcal{C}(\phi) \wedge P \\
\mathcal{C}(\phi, i = j) & = & \mathcal{C}(\phi) \wedge (i = j)
\end{array}$$

$\mathcal{C}$ is well-defined: Assuming a straightforward weight-function for sort definitions, a termination order for $\mathcal{C}(\phi)$ can be given as the lexicographic order of the sum of the weights of all sort definitions in ϕ and the length of ϕ.

The following lemma expresses a useful correspondence between ϕ and $\mathcal{C}(\phi)$:

Lemma 13
Let ϕ and ϕ' be index contexts and θ an index substitution such that $\mathbf{dom}(\theta) = \mathbf{dom}(\phi')$. Then

$$\phi \vdash \theta : \phi' \quad \textit{iff} \quad \phi \models \mathcal{C}(\phi')[\theta].$$

Proof: We conduct the proof by structural induction over ϕ. In the base case, $\phi = \cdot$, we have $\phi \vdash [] : \cdot$ and $\phi \models \top[\theta]$ for all substitutions θ. Both directions follow immediately (using $\mathbf{dom}(\theta) = \mathbf{dom}(\phi')$ from right to left).

A non-empty index context has form $\phi, a : \gamma$ or ϕ, P. We examine the first of these cases, the second case follows by similar reasoning.

We have to show that

$$\phi \vdash \theta[a \mapsto i] : \phi', a : \gamma \quad \text{iff} \quad \phi \models \mathcal{C}(\phi', a : \gamma)[\theta[a \mapsto i]],$$

which is equivalent to

$$(\phi \vdash \theta : \phi' \ \text{ and } \ \phi \vdash i : \gamma[\theta]) \quad \text{iff} \quad (\phi \models \mathcal{C}(\phi')[\theta] \ \text{ and } \ \phi \models \mathcal{C}(a : \gamma)[\theta[a \mapsto i]]).$$

Assuming that for all i and γ

$$\phi \vdash i : \gamma \quad \text{iff} \quad \phi \models \mathcal{C}(a : \gamma)[a \mapsto i], \qquad (*)$$

we can conclude the proof using the induction hypothesis.

[7]Because of the way $\mathcal{C}$ handles product sorts, it actually is defined on contexts that assign sorts to index objects rather than only index variables.

It remains to show Equation (*). We use induction over the structure of γ and demonstrate the case of a sort definition of form $\{k : \gamma \mid P\}$: We have to show that

$$\phi \vdash i : \{k : \gamma \mid P\} \quad \text{iff} \quad \phi \models \mathcal{C}(a : \{k : \gamma \mid P\})[a \mapsto i]$$

Assuming the left-hand side, it follows with rule-inversion that $\phi \vdash i : \gamma$ and $\phi \models P[k \mapsto i]$. Using the induction hypothesis, we further can derive that $\phi \models \mathcal{C}(a : \gamma)[a \mapsto i]$, so obviously $\phi \models (\mathcal{C}(a : \gamma \wedge P[k \mapsto a]))[a \mapsto i]$, which is equivalent to $\phi \models (\mathcal{C}(a : \{k : \gamma \mid P\}))[a \mapsto i]$. The direction from right to left follows with similar reasoning steps. □

B.4 Extraction of recurrence equations—correctness

In this section, we present a proof of the correctness of the extraction algorithm for recurrence equations, as stated in Theorem 4 on page 29. As the theorem is in three parts, we divide the proof into three parts.

B.4.1 The result of extraction is well-typed

The first part of Theorem 4 on page 29 follows directly from a lemma that relates the type derivation of a function body to the type derivation of the extracted recurrence-equation term.

Lemma 14
If $\phi; \Gamma \vdash e : \rho \blacktriangleright t$ is derivable for an expression e, then $\widetilde{\phi}; \widetilde{\Gamma} \vdash t : \mathbf{C}$ is derivable.

Proof: The proof is conducted by structural induction over the type derivation of e. In case the last rule of the type derivation is ty-eq, then the lemma follows by induction hypothesis. Because the remaining typing rules for $\mathrm{DML}_0^{\Pi}(C)$ are syntax directed, we proceed by examining the different syntactic forms of e as given by the grammar in Figure 8 on page 19. We show the case of a function call $F\ [i_0] \dots [i_k]\ e$; all other cases can be shown in a similar way.

For a function call $F[i_0] \dots [i_k](e)$, the extracted recurrence-equation term is $t := t' + \mathbf{c}_F + (F^c\ i_0 \dots i_k)$ where t' has been extracted from e. The last rule of the type derivation must be ty-app, i.e., for some ρ_1, ρ_2 there exist derivations

$$\phi; \Gamma \vdash e : \rho_1 \tag{1}$$
$$\phi; \Gamma \vdash F[i_0] \dots [i_k] : \rho_1 \rightarrow \rho_2 \tag{2}$$

Analyzing the shape of a type derivation for t, we see that we need to show

$$\widetilde{\phi}; \widetilde{\Gamma} \vdash t' : \mathbf{C} \tag{1$'$}$$
$$\widetilde{\phi} \vdash i_0 : \widetilde{\gamma_0} \quad \dots \quad \widetilde{\phi} \vdash i_k : \widetilde{\gamma_k}, \tag{2$'$}$$

assuming that $\Gamma(F) = \Pi\, a_0 : \gamma_0 \ldots \Pi\, a_k : \gamma_k \,.\, \rho_1' \to \rho_2'$. From (1), we can show (1′) using the induction hypothesis. From (2), we can derive (2′): With a straightforward induction on k one can show that (2) implies

$$\phi \vdash [a_0 \mapsto i_0, \ldots, a_k \mapsto i_k] : (a_0 : \gamma_0, \ldots, a_k : \gamma_k).$$

With Lemma 12 on page 43, it follows that

$$\widetilde{\phi} \vdash [a_0 \mapsto i_0, \ldots, a_k \mapsto i_k] : (a_0 : \widetilde{\gamma_0}, \ldots, a_k : \widetilde{\gamma_k}),$$

which, because there are no dependencies between two index types $\widetilde{\gamma_l}$ and $\widetilde{\gamma_{l'}}$, implies (2′). □

B.4.2 The result of extraction is a cost bound

We need to show that the cost of executing $F\ [\vec{\imath}]\ v$ for some user-defined function F and some value v is bounded by $F^c\ \vec{\imath}$, where F^c is the cost recurrence extracted for F. We start with Lemma 15, which says that when extracting a recurrence-equation term t from a DML expression e, then t defines a bound for e under the assumption that all calls to user-defined functions in e are bounded correctly by the corresponding calls to F^c in t.

Lemma 15
Let $\phi; \Gamma_F, \Gamma \vdash e : \rho \blacktriangleright t$ be derivable. Let θ be a substitution such that θ_ϕ substitutes index variables for ground terms, θ_Γ substitutes (normal) variables for values, and $\cdot; \Gamma_F \vdash \theta : (\phi; \Gamma)$ is derivable, i.e., θ is type-sound with respect to ϕ and Γ. Let Θ be an environment such that $\vdash \Theta : \Gamma_F$ is derivable. Let Ψ be a mapping from $\mathbf{dom}(\Gamma_F)$ to functions such that

1. *if $\Gamma_F(F) = \Pi\, a_0 : \gamma_0 \ldots \Pi\, a_l : \gamma_l \,.\, \rho_1 \to \rho_2$ then*

$$\Psi(F^c) \in [\mathcal{I}[\![\widetilde{\gamma_0}]\!] \to \ldots \to \mathcal{I}[\![\widetilde{\gamma_l}]\!] \to \mathbf{C}_\perp]$$

2. *if $F[\vec{\imath}](u)$ type-checks under Γ_F and $\mathcal{T}[\![F^c\ \vec{\imath}]\!]\Psi[] = \llcorner z \lrcorner$ then*

$$F[\vec{\imath}](u) \longrightarrow_{\Theta^*} \langle u', z' \rangle \quad \text{with } z' \leq z.$$

Then if $\mathcal{T}[\![t]\!]\Psi(\mathcal{I}[\![\theta_\phi]\!]) = \llcorner z \lrcorner$, then $e^*[\theta] \longrightarrow_{\Theta^*} \langle v', z' \rangle$ with $z' \leq z$.

Proof: We use structural induction over the type derivation of e. In case the last rule of the type derivation is `ty-eq`, then the lemma follows by induction hypothesis. Because the remaining typing rules for $\mathrm{DML}_0^\Pi(C)$ are syntax directed, we proceed by examining the different syntactic forms of e as given by the grammar in Figure 8 on page 19. We show two interesting cases: function application and case expression.

For a function application $F[\vec{\imath}](e)$, the extracted recurrence-equation term is $t + \mathbf{c}_F + F^c\ \vec{\imath}$, where t has been extracted from e. We express $(F[\vec{\imath}](e))^*$, (see Figure 9 on page 21 for the definition of the monadic translation $(\cdot)^*$) as a DML program by removing syntactic sugar introduced in the translation (see Figure 10 on page 21) and examine the shape of a possible derivation for

$$(F[\vec{\imath}](e))^*[\theta] \longrightarrow_{\Theta^*} \langle v', z' \rangle.$$

It is easy to see that such a derivation can exist only if there are $z'_F, z'_e \in \mathbf{C}$ such that $z' = z'_F + \mathbf{c}_F + z'_e$ and the following two assumptions hold:

(1) For some value v_e, there is a derivation $e^*[\theta] \longrightarrow_{\Theta^*} \langle v_e, z'_e \rangle$.

(2) There is a derivation $F\ [\vec{\imath}[\theta]]\ v_e \longrightarrow_{\Theta^*} \langle v', z'_F \rangle$.

Assume now that $\mathcal{T}[\![t + \mathbf{c}_F + F^c\ \vec{\imath}]\!]\Psi(\mathcal{I}[\![\theta_\phi]\!]) = \llcorner z \lrcorner$; using the semantics definition from Figure 13 on page 24 the following three facts follow easily:

(1′) For some $z_e \in \mathbf{C}$ we have $\mathcal{T}[\![t]\!]\Psi(\mathcal{I}[\![\theta_\phi]\!]) = \llcorner z_e \lrcorner$.

(2′) For some $z_F \in \mathbf{C}$, we have $\mathcal{T}[\![F^c\ \vec{\imath}]\!]\Psi(\mathcal{I}[\![\theta_\phi]\!]) = \llcorner z_F \lrcorner$.

(3′) We have $z = z_F + \mathbf{c}_F + z_e$.

Using the induction hypothesis and (1′), we can deduce (1) for some $z'_e \leq z_e$. From (2′), it follows by assumption that (2) holds for some $z'_F \leq z_F$; this is because $\mathcal{T}[\![F^c\ \vec{\imath}]\!]\Psi(\mathcal{I}[\![\theta_\phi]\!])$ is equivalent to $\mathcal{T}[\![F^c\ (\vec{\imath}[\theta_\phi])]\!]\Psi[]$. Finally, using (3′) we can infer that $z' \leq z$, which concludes the proof for this case.

For a case expression of form (**case** e **of** $p_0 \Rightarrow e_0 \mid \ldots p_k \Rightarrow e_k$), the extracted recurrence-equation term is

$$t + (\mathbf{cond}\ br_0 \mid \ldots \mid br_k)$$

where t is extracted from e by

$$\phi; \Gamma_F, \Gamma \vdash e : \rho' \ \blacktriangleright\ t$$

and every br_j from the type-derivation of a branch $(p_j \Rightarrow e_j)$ as

$$\Phi_{j1} \to \forall(\mathbf{dom}(\phi'_j)\backslash\mathbf{dom}(\theta_j)).\Phi_{j2} \to t[\theta_j]$$

where

$$\begin{array}{l} p \downarrow \rho' \ \rhd\ (\phi'_j; \Gamma'_j) \\ \phi, \phi'_j; \Gamma, \Gamma'_j \vdash e_j : \rho \ \blacktriangleright\ t_j \\ \Phi_{j1} = \exists(\mathbf{dom}(\phi, \phi'_j)\backslash\mathbf{var}(\rho')).\mathcal{C}(\phi, \phi'_j) \\ \theta_j = \mathit{mk\_subst}_{\mathbf{dom}(\phi'_j)}(\mathcal{C}(\phi'_j)) \\ \Phi_{j2} = \exists(\mathbf{dom}(\theta_j)).\mathcal{C}(\phi'_j) \end{array}$$

Like before, we examine the shape of a possible derivation for

$$(\mathbf{case}\ e\ \mathbf{of}\ p_0 \Rightarrow e_0 \mid \ldots p_k \Rightarrow e_k)^*[\theta] \longrightarrow_{\Theta^*} \langle v', z' \rangle,$$

and see that such a derivation can exist only if there are $z'_e, z'_b \in \mathbf{C}$ such that $z' = z'_e + z'_b$ and the following two assumptions hold:

(1) For some value v_e, there is a derivation $e^*[\theta] \longrightarrow_{\Theta^*} \langle v_e, z'_e \rangle$.

(2) For some j with $0 \leq j \leq k$ for which $\mathbf{match}(p_j, v_e) \Longrightarrow \theta'$ is derivable, there is a derivation $e_j{}^*[\theta \circ \theta'] \longrightarrow_{\Theta^*} \langle v', z'_b \rangle$.

Assume now that $\mathcal{T}[\![t + (\mathbf{cond}\ br_0 \mid \ldots \mid br_k)]\!]\Psi(\mathcal{I}[\![\theta_\phi]\!]) = \llcorner z \lrcorner$; using the semantics definition from Figure 13 on page 24 we can deduce the following facts:

(1′) For some $z_e \in \mathbf{C}$ we have $\mathcal{T}[\![t]\!]\Psi(\mathcal{I}[\![\theta_\phi]\!]) = \llcorner z_e \lrcorner$.

(2′) There is some $z_b \in \mathbf{C}$ such that for every branch br_j

$$\Phi_{j1} \rightarrow \forall(\mathbf{dom}(\phi'_j)\backslash\mathbf{dom}(\theta_j)).\Phi_{j2} \rightarrow t[\theta_j]$$

we have that if $\models \Phi_{j1}[\theta_\phi]$ then for all $\vec{z}$ with $\models \Phi_{j2}[\theta_\phi[\vec{a} \mapsto \vec{z}]]$ there is $z_B \leq z_b$ such that $\mathcal{T}[\![t_j[\theta_j]]\!]\Psi(\mathcal{I}[\![\theta_\phi[\vec{a} \mapsto \vec{z}]]\!]) = \llcorner z_B \lrcorner$

(3′) We have $z = z_e + z_b$.

Using the induction hypothesis and (1′), we can deduce (1) for some $z'_e \leq z_e$. In the following, we prove (2), using the induction hypothesis and (2′):

- We first show that whenever $\mathbf{match}(p_j, v_e) \Longrightarrow \theta'$ is derivable, then $\models \Phi_{j1}[\theta_\phi]$, i.e., the precondition for (2') holds.

  We examine the form of $\Phi_{j1} = \exists\vec{b}, \vec{c}.\mathcal{C}(\phi, \phi'_j)$, where $\vec{b}$ are the variables in $\mathbf{dom}(\phi'_j)$ and $\vec{c}$ are the variables in $\mathbf{dom}(\phi)\backslash\mathbf{var}(\rho')$. We need to show

  $$\models (\exists\vec{b}, \vec{c}.\mathcal{C}(\phi, \phi'_j))[\theta_\phi].$$

  In fact, θ_ϕ and $\theta'{}_\phi$ hold witnesses for $\vec{b}$ and $\vec{c}$, respectively: By assumption, we have $\cdot; \Gamma_F \vdash \theta : (\phi; \Gamma)$, which implies $\vdash \theta_\phi : \phi$. Further, with the first part of Theorem 7 on page 35, it follows that $\phi \vdash \theta_{j_\phi} : \phi'$. Using Lemma 13 on page 44, we derive $\models \mathcal{C}(\phi)[\theta_\phi]$ and $\phi \models \mathcal{C}(\phi'_j)[\theta'{}_\phi]$. With Lemma 5 on page 33, we then can derive $\models (\mathcal{C}(\phi'_j)[\theta'{}_\phi])[\theta_\phi]$, so all in all we have we have $\models \mathcal{C}(\phi, \phi'_j)[\theta'{}_\phi\ \theta_\phi]$, which implies $\models (\exists\vec{b}, \vec{c}.\mathcal{C}(\phi, \phi'_j))[\theta_\phi]$.

- With similar reasoning, we show that whenever $\mathbf{match}(p_j, v_e) \Longrightarrow \theta'$ is derivable, then $\models \Phi_{j2}[\theta_\phi[\vec{a} \mapsto \theta'{}_\phi(\vec{a})]]$ (where $\vec{a} := \mathbf{dom}(\phi'_j)\backslash\mathbf{dom}(\theta_j)$) holds, so with (2′) it follows that

  $$\mathcal{T}[\![t_j[\theta_j]]\!]\Psi(\mathcal{I}[\![\theta[\vec{a} \mapsto \theta'{}_\phi(\vec{a})]]\!]) = \llcorner z_B \lrcorner \quad \text{for some } z_B \leq z_b. \qquad (*)$$

- Knowing (*), we can show (2) by induction hypothesis if

  $$\mathcal{T}[\![t_j[\theta_j]]\!]\Psi(\mathcal{I}[\![\theta_\phi[\vec{a} \mapsto \theta'{}_\phi(\vec{a})]]\!]) = \mathcal{T}[\![t_j]\!]\Psi(\mathcal{I}[\![\theta_\phi \circ \theta'{}_\phi]\!]).$$

  With a straightforward structural induction over t_j we can show that

  $$\mathcal{T}[\![t_j[\theta_j]]\!]\Psi(\mathcal{I}[\![\theta_\phi[\vec{a} \mapsto \theta'{}_\phi(\vec{a})]]\!]) = \mathcal{T}[\![t_j]\!]\Psi(\mathcal{I}[\![\theta_j \circ \theta_\phi[\vec{a} \mapsto \theta'{}_\phi(\vec{a})]]\!])$$

It remains to show that $b[\theta_j \circ \theta_\phi[\vec{a} \mapsto \theta'_\phi(\vec{a})]] = b[\theta_\phi \circ \theta'_\phi]$ for all $b \in \mathbf{dom}(\theta \circ \theta')$. Some reasoning about the domains of the comprised substitutions allows us to rearrange their composition and show instead that

$$b[(\theta_j \circ [\vec{a} \mapsto \theta'_\phi(\vec{a})])\theta_\phi] = b[\theta'_\phi\ \theta_\phi] \qquad (**)$$

For $b \in \vec{a}$ and $b \in \mathbf{dom}(\theta)$ there is nothing to show. Consider now $b \in \mathbf{dom}(\theta_j)$. By definition of θ_j we know that $\phi' \models b = b[\theta_j]$. Using Lemma 5 on page 33 on $\vdash \theta_\phi : \phi$ and $\phi \vdash \theta_{j_\phi} : \phi'_j$ (established above), we can derive that $b[\theta_j \circ \theta'_\phi \circ \theta_\phi] = b[\theta'_\phi \circ \theta_\phi]$. Equation (**) follows with some basic reasoning about substitutions.

Now, with ($3'$) and the fact that $z_B \leq z_b$, it follows that $z' \leq z$, which concludes the proof for this case. □

We have shown that extracting a recurrence-equation term t from a DML expression e yields a valid bound under the assumption that we have a valid bound F^c for every user-defined function F called in e. We now show that the semantics of a recurrence equation G^c, which is based on the recurrence-equation term extracted from the body of a user-defined function G (see Section 4.3.2), indeed defines a bound for G.

Lemma 16

Let ($\mathbf{fix}\, G : \Pi\, \vec{a} : \vec{\gamma}\, .\, \rho_1 \to \rho_2 . \mathbf{lam}\ x : \rho_1\, .\, \mathbf{case}\ x\ \mathbf{of}\ \langle x_0, \ldots, x_k\rangle \Rightarrow e$) *be typable under* Γ_F *and let*

$$\cdot ; \Gamma_F, G : \Pi\, \vec{a} : \vec{\gamma}\, .\, \rho_1 \to \rho_2, x : \rho_1, x_0 : \rho_1^0, \ldots, x_k : \rho_1^k \vdash e : \rho_2 \ \blacktriangleright\ t$$

be derivable. Let Θ *be an environment such that* $\vdash \Theta : \Gamma_F$ *is derivable. Let* Ψ *be a mapping from* $\mathbf{dom}(\Gamma_F)$ *to functions such that*

1. *if* $\Gamma_F(F) = \Pi\, a_0 : \gamma_0 \ldots \Pi\, a_l : \gamma_l\, .\, \rho_1 \to \rho_2$ *then*
$$\Psi(F^c) \in [\mathcal{I}[\![\widetilde{\gamma_0}]\!] \to \ldots \to \mathcal{I}[\![\widetilde{\gamma_l}]\!] \to \mathbf{C}_\perp]$$

2. *if* $F[\vec{\imath}](u)$ *type-checks under* Γ_F *and* $\mathcal{T}[\![F^c\ \vec{\imath}]\!]\Psi[] = \llcorner z \lrcorner$ *then*
$$F[\vec{\imath}](u) \longrightarrow_{\Theta^*} \langle u', z'\rangle \quad \textit{with } z' \leq z.$$

Let further

$$\begin{array}{rcl} \varphi & := & \mathit{fix}(\lambda\mathcal{F}.\lambda\vec{n}.\mathcal{T}[\![t]\!](\Psi[G^c \mapsto \mathcal{F}])[\vec{a} \mapsto \vec{n}]) \\ \Theta_1 & := & \Theta[G \mapsto \mathbf{lam}\ x : \rho_1\, .\, \mathbf{case}\ x\ \mathbf{of}\ \langle x_0, \ldots, x_k\rangle \Rightarrow e] \end{array}$$

Then, if $G[\vec{\imath}](u)$ *type-checks under* $\Gamma_F, G : \Pi\, \vec{a} : \vec{\gamma}\, .\, \rho_1 \to \rho_2$ *and*

$$\mathcal{T}[\![G^c\ [\vec{\imath}]]\!](\Psi[G^c \mapsto \varphi])[] = \llcorner z \lrcorner,$$

we have $G[\vec{\imath}](u) \longrightarrow_{\Theta_1{}^*} \langle v', z'\rangle$ *with* $z' \leq z$.

Proof: We use fixed-point induction over the definition of φ. For the base case, we have

$$\mathcal{T}[\![G\ [\vec{\imath}]]\!](\Psi[G^c \mapsto \varphi])[] = \bot,$$

so the lemma is vacuously true. For the induction step, assume that for φ', if $G[\vec{\imath}](u)$ type-checks under $\Gamma_F, G : \Pi\, \vec{a} : \vec{\gamma}\,.\,\rho_1 \to \rho_2$ and

$$\mathcal{T}[\![G^c\ [\vec{\imath}]]\!](\Psi[G^c \mapsto \varphi'])[] = \llcorner z \lrcorner,$$

we have $G[\vec{\imath}](u) \longrightarrow_{\Theta_1{}^*} \langle v', z' \rangle$ with $z' \leq z$. We have to show that if

$$\mathcal{T}[\![G\ [\vec{\imath}]]\!](\Psi[G^c \mapsto \lambda\vec{n}.\mathcal{T}[\![t]\!](\Psi[G^c \mapsto \varphi'])[\vec{a} \mapsto \vec{n}]])[] = \llcorner z \lrcorner \tag{1}$$

then

$$G[\vec{\imath}]\ u \longrightarrow_{\Theta_1{}^*} \langle v', z' \rangle \tag{2}$$

with $z' \leq z$.

Using the definitions of the denotational semantics (Figure 13 on page 24) of recurrence equations and operational semantics of DML (Figure 20 on page 39), we see that Equations (1) and (2) are equivalent to

$$\mathcal{T}[\![t]\!](\Psi[G^c \mapsto \varphi'])[\vec{a} \mapsto \mathcal{I}[\![\vec{\imath}]\!]] = \llcorner z \lrcorner \tag{1'}$$

$$e^*[\vec{a} \mapsto \vec{\imath}][x \mapsto u][x_0 \mapsto \mathbf{fst}(u)] \ldots [x_k \mapsto \mathbf{snd}^k(u)] \longrightarrow_{\Theta_1{}^*} \langle v', z' \rangle \tag{2'}$$

We can use Lemma 15 on page 46; its preconditions are satisfied:

- From the fact that $G[\vec{\imath}](u)$ type-checks under $\Gamma_F, G : \Pi\, \vec{a} : \vec{\gamma}\,.\,\rho_1 \to \rho_2$, we can deduce that the substitution

  $$[\vec{a} \mapsto \vec{\imath}][x \mapsto u][x_0 \mapsto \mathbf{fst}(u)] \ldots [x_k \mapsto \mathbf{snd}^k(u)]$$

  is type-sound with respect to $\vec{a} : \vec{\gamma}$ and $x : \rho_1, x_0 : \rho_1^0, \ldots, x_k : \rho_1^k$.
- The environment $(\Psi[G^c \mapsto \varphi'])$ has the required properties, for $F^c \in \mathbf{dom}(\Theta)$ by assumption, and for G^c by induction hypothesis.

□

Part 2 of Theorem 4 on page 29 follows from Lemma 16 on the preceding page with a straightforward induction proof over the number of user-defined functions in program p.

B.4.3 Result of extraction is a recurrence

The intuition behind the test described in Section 4.4.3 of whether the extracted cost bound is a recurrence is to use the constraint information contained in $\mathrm{DML}_0^{\Pi}(C)$ type derivations for showing that the argument on each recursive call decreases. Without further proof, however, such an argument only shows the termination of the examined DML program. We further need to argue that conclusions drawn from constraint information about index arguments to recursive function calls also holds for arguments to the corresponding calls occurring within an occurrence equation:

Theorem 17
Assume that the extraction algorithm annotates every call $F^c\ \vec{\imath}$ with the index context it was extracted under, i.e., $\phi'; \Gamma' \vdash F[\vec{\imath}](e') : \rho'$ gives rise to $F^{c,\phi'}\vec{\imath}$, and that the semantics $\mathcal{T}[\![\cdot]\!]$ ignores such annotations. If $\phi; \Gamma \vdash e : \rho \blacktriangleright t$ and $\vdash \theta : \phi$, then, in the unfolded definition of $\mathcal{T}[\![t]\!]\Psi\theta$, for all applications $\Psi(F^{c,\phi'})\ \mathcal{I}[\![\vec{\imath}[\theta']]\!]$ we have $\vdash \theta' : \phi'$.

Proof: As in the proof of Lemma 15 on page 46, we use structural induction over the type derivation of e. The only interesting case is that of a case expression **case** e **of** $p_0 \Rightarrow e_0 \mid \ldots p_k \Rightarrow e_k$. The extracted recurrence-equation term is

$$t + (\textbf{cond}\ br_0 \mid \ldots \mid br_k)$$

where t is extracted from e by

$$\phi; \Gamma_F, \Gamma \vdash e : \rho' \blacktriangleright t$$

and every br_j from the type-derivation of a branch $(p_j \Rightarrow e_j)$ as

$$\Phi_{j1} \to \forall(\textbf{dom}(\phi'_j) \backslash \textbf{dom}(\theta_j)).\Phi_{j2} \to t[\theta_j]$$

where

$$\begin{array}{l} p \downarrow \rho' \ \rhd\ (\phi'_j; \Gamma'_j) \\ \phi, \phi'_j; \Gamma, \Gamma'_j \vdash e_j : \rho \blacktriangleright t_j \\ \Phi_{j1} = \exists(\textbf{dom}(\phi, \phi'_j) \backslash \textbf{var}(\rho')).\mathcal{C}(\phi, \phi'_j) \\ \theta_j = mk\_subst_{\textbf{dom}(\phi'_j)}(\mathcal{C}(\phi'_j)) \\ \Phi_{j2} = \exists(\textbf{dom}(\theta_j)).\mathcal{C}(\phi'_j) \end{array}$$

For $\mathcal{T}[\![t]\!]\Psi\theta$ we can simply use the induction hypothesis. For the conditional, however, we have to examine all possible calculations $\mathcal{T}[\![t_j[\theta_j]]\!]\Psi(\theta[\vec{a} \mapsto \vec{z}])$ (with $\vec{a} := \textbf{dom}(\phi'_j) \backslash \textbf{dom}(\theta_j)$). With a straightforward structural induction on t_j one can show that

$$\mathcal{T}[\![t_j[\theta_j]]\!]\Psi(\theta[\vec{a} \mapsto \vec{z}]) = \mathcal{T}[\![t_j]\!]\Psi(\theta_j \circ (\theta[\vec{a} \mapsto \vec{z}])).$$

To use the induction hypothesis, it remains to show that $\vdash (\theta_j \circ \theta[\vec{a} \mapsto \vec{z}]) : (\phi, \phi'_j)$, which follows with a short derivation using Lemma 13 on page 44 and the assumptions about ϕ, ϕ'_j, θ, θ_j, and $\vec{z}$. □

Theorem 17 shows that index-based reasoning about the index arguments to functions in a DML program carry over to the corresponding recurrence equation, hence the third part of Theorem 4 on page 29 holds.