Efficient Algorithms for Isomorphisms of Simple Types

Yoav Zibin
Technion—Israel Institute of Technology
zyoav@cs.technion.ac.il

Joseph (Yossi) Gil
Technion—Israel Institute of Technology
yogi@cs.technion.ac.il

Jeffrey Considine
Boston University
jconsidi@cs.bu.edu

ABSTRACT

The *first order isomorphism problem* is to decide whether two non-recursive types using product- and function-type constructors, are isomorphic under the axioms of commutative and associative products, and currying and distributivity of functions over products. We show that this problem can be solved in $O(n \log^2 n)$ time and $O(n)$ space, where n is the input size. This result improves upon the $O(n^2 \log n)$ time and $O(n^2)$ space bounds of the best previous algorithm. We also describe an $O(n)$ time algorithm for the *linear isomorphism problem*, which does not include the distributive axiom, whereby improving upon the $O(n \log n)$ time of the best previous algorithm for this problem.

Categories and Subject Descriptors

F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Lambda calculus and related systems*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Type structure*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Data types and structures*; G.4 [**Mathematical Software**]: Algorithm design and analysis

General Terms

Algorithms, Design, Languages, Theory

Keywords

First order isomorphism, Linear isomorphism, Non-recursive types, Simple types, Type signature

1. Introduction

It is a matter of basic high school algebra to prove the equality

$$\left((\mathsf{ab})^{(\mathsf{a}^{\mathsf{b}})}\right)^{(\mathsf{b}^{\alpha})} = \mathsf{a}^{\mathsf{a}^{\mathsf{b}}\mathsf{b}^{\mathsf{a}}}\mathsf{b}^{\mathsf{b}^{\mathsf{a}}\mathsf{a}^{\mathsf{b}}}. \qquad (1.1)$$

Yet, as we shall see in this paper, a systematic and efficient production of such a proof is non-trivial. With the familiar perspective of viewing multiplication as product-types, exponentiation as function-types, and variables as primitive-types, (1.1) becomes an instance of a simple, i.e., non-recursive, type isomorphism problem. In its turn, type isomorphism has close connections to category theory [5] and intuitionistic logic [13].

The isomorphism variant which concerns us here is characterized by commutativity and associativity of products, and currying and distributivity of functions over products. This variant has practical interest in the context of the search for compatible functions in function libraries.[1] (A detailed treatise of this application can be found in Di Cosmo's book [9], which discusses also extensions to second order types and the ML type theory.)

More formally, we consider the set of first order isomorphisms holding in all models of the lambda calculus with product-types (surjective pairing), function-types, and unit types, as defined by the following grammar

$$\tau \quad ::= \quad \mathbf{1} \quad | \quad x \quad | \quad \tau \to \tau \quad | \quad \tau \times \tau \quad ,$$

where $\mathbf{1}$ is the unit type, x stands for an arbitrary primitive-type, $\to$ denotes a function-type, and $\times$ denotes a product-type.

These isomorphisms can be derived from the following seven axioms schemas.

$$\begin{aligned}
A \times \mathbf{1} &= A \\
A \to \mathbf{1} &= \mathbf{1} \\
\mathbf{1} \to A &= A \\
A \times B &= B \times A && \text{(Commutative)} \\
A \times (B \times C) &= (A \times B) \times C && \text{(Associative)} \\
(A \times B) \to C &= A \to (B \to C) && \text{(Currying)} \\
A \to (B \times C) &= (A \to B) \times (A \to C) && \text{(Distributive)}
\end{aligned}$$

(Here and henceforth, the range of variables A, B and C is any type expression.)

For a long time, the problem of deciding first order isomorphisms of simple types was thought to require exponential time [5]. It was recently shown [7] that the variant of our interest can be decided in $O(n^2 \log n)$ time and $O(n^2)$ space, where n is the length of some standard representation of the two input types. The main contribution of this paper is an improvement of this result to

$$O(n \log^2 n)$$

time and $O(n)$ space. We also give algorithms using $O(n)$ time and space for important special cases.

1.1 Background

The arithmetic version of these seven axioms (substituting multiplication, exponentiation, and the constant one, for $\times$, $\to$ and $\mathbf{1}$) was proved to be complete for the Cartesian closed categories [5,

[1] Besides being sufficient for the proof of equations such as (1.1).

POPL'03, January 15–17, 2003, New Orleans, Louisiana, USA.

18]. Since the models of the lambda calculus with unit, product- and function-types are exactly the Cartesian closed categories [5], the set is also complete for the type isomorphisms we examine. Through the Curry-Howard isomorphism [13], these isomorphisms are also equivalent to equational equality in positive intuitionistic logic so the same axioms apply there too (again, with appropriate notational changes).

Besides their theoretical connections, type isomorphisms can be used as a means of searching large program libraries. Specifically, the desired type of a function is used as a search key and functions with isomorphic types are returned as candidates. A famous example [17] shows that even the simple function, folding a list, can be implemented with many different types, varying argument order and the use of "Curried" style. Employing type isomorphisms in the search will retrieve all compatible function implementations. Moreover, the isomorphism proof can often automatically generate bridge code converting the functions found to the desired type.

Second order isomorphisms augment first order isomorphisms with universal quantifiers, as in $\forall A.A \to A = \forall B.B \to B$. Universal quantifiers make second order isomorphisms more effective in searching program libraries since they are necessary to capture parametric polymorphism. While some of the issues of second order isomorphisms are similar (some of the space sharing techniques are applicable), they are known to be graph isomorphism complete [4, 9] and we do not attempt to decide them in this work. A different system of type isomorphisms is that of the core ML language. It is known [8] that second order isomorphisms are insufficient to describe these, although the addition of one more axiom suffices.

More recently, the Mockingbird project has renewed interest in such searches when using recursive types [3]. One challenge still remaining is to find a consistent scheme, as the first one considered was later shown to be inconsistent [2, 15].

Other variants of the non-recursive type isomorphism problem were considered in the literature. For example, Gil [11] describes how algorithms for polynomial equality can be used for deciding isomorphism in the "polynomial type system", i.e., the system comprising union and product-types, as well as the *arithmetical rules*, i.e., commutative, associative rules for product and union, and distributivity of product over union.

The more general isomorphism problem, for a non-recursive type system which includes product, union *and* function-types along with the arithmetical rules is equivalent to Tarski's *high school algebra problem* [19]. Such a system does not have a finite and complete set of axioms. Nonetheless, there exists a (non-polynomial) algorithm for determining isomorphism [12]. There also exists a (non-polynomial) algorithm for deciding isomorphism in the "algebraic type system", i.e., recursive types along with the arithmetical rules [11]. Finally, we should mention that adding empty and sum types breaks down the relationship between the equational theory and type isomorphisms [10].

1.2 First Order Isomorphism and Variants

In this paper, we concentrate on first order isomorphism (the following Def. 1.1) and two restricted variants (Def. 1.2 and Def. 1.3 below):

DEFINITION 1.1 (FIRST ORDER ISOMORPHISM).
The first order isomorphism problem is to decide whether two types are equal under a theory of equality plus the above seven axiom schemas.

Deciding first order isomorphisms of simple types has been known to be decidable for over a decade [5]. The theory describing these isomorphisms is often referred to as $Th^1_{\times T}$ [5,6,9]. Previous to this work, the best known bound was $O(n^2 \log n)$ time using $O(n^2)$ space [7]. *Our main result is in reducing the time to $O(n \log^2 n)$ time and the space to $O(n)$.*

DEFINITION 1.2 (PRODUCT ISOMORPHISM).
The product isomorphism problem is to decide whether two types are equal under a theory of equality plus the first five axiom schemas above (that is, all but the Currying and Distributive laws).

When the Commutative and Associative axioms apply, we write products without parenthesis, e.g.,

$$\mathsf{abracadabra}. \tag{1.2}$$

(Lower case, sanserif letters denote here and henceforth primitive-types. We shall use the arithmetical and type notations interchangeably. No confusion will arise.) An instance of this problem variant is to determine whether the above is isomorphic to

$$\mathsf{carrabadaba}. \tag{1.3}$$

One may be tempted to attack the problem by bringing each product into a unique sorted normal form, which in this case is

$$\mathsf{aaaaabbcdrr}. \tag{1.4}$$

In this paper we show that (non-recursive) product isomorphism is decidable in linear time.[2] This result is based on the observation that it can be determined that (1.2) and (1.3) are isomorphic without a super-linear sorting procedure, but rather an algorithm for *multi-set comparison*. More generally, to determine whether $\prod_{i=1}^{k} A_i$ is isomorphic to $\prod_{i=1}^{k} B_k$ the multi-set comparison algorithm checks whether there exists a permutation π such that $A_{\pi(i)}$ is isomorphic to B_i.

This product isomorphism variant was not considered previously as such in the literature. Palsberg and Zhao [15] gave an $O(n^2)$ time algorithm for *recursive* product isomorphism, defined by the addition of a grammar rule

$$\tau ::= \mu\alpha.\tau$$

where α is a type variable, and the folding/unfolding axiom

$$\mu\alpha.A = A[(\mu\alpha.A)/\alpha].$$

(As usual, the notation $A[B/\alpha]$ stands for a the type expression A where each occurrence of α is replaced by B.) Isomorphism between recursive product-types are defined in terms of their infinite unfoldings which are regular trees. This result was later improved to $O(n \log n)$ time [14] using a reduction to the problem of finding size-stable partitions of a directed graph.

DEFINITION 1.3 (LINEAR ISOMORPHISM).
The linear isomorphism problem is to decide whether two types are equal under a theory of equality plus the first six axiom schemas above (that is, all but the Distributive law).

Polynomial time results for this problem were known before those of the first order problem. Linear isomorphism can be decided in linear space and $O(n \log^2 n)$ time [1]. Although not previously mentioned, both algorithms [7, 14] improve the running time to $O(n \log n)$. *We show that linear isomorphism is also decidable in linear time.*

[2] Jha (personal communication, September 2002) reports on independent discovery of an algorithm for the this sub-problem, with similar complexity bounds.

Linear isomorphism combined with the unfolding rule cannot be treated by algorithms for recursive type isomorphisms, since the application of this rule may produce infinite products, as in, e.g.,

$$\mu\alpha.(\mathsf{a} \to \alpha).$$

1.3 Reduction systems and Normal forms

Isomorphism proofs are usually based upon reduction systems producing a normal form representation of the input, which can be more easily compared. For example, the reduction system of Rittri [17] has seven rules

$$\begin{array}{lrcl} \mathcal{R}.1 & A \times \mathbf{1} & \Rightarrow & A \\ \mathcal{R}.2 & \mathbf{1} \times A & \Rightarrow & A \\ \mathcal{R}.3 & A \to \mathbf{1} & \Rightarrow & \mathbf{1} \\ \mathcal{R}.4 & \mathbf{1} \to A & \Rightarrow & A \\ \mathcal{R}.5 & A \times (B \times C) & \Rightarrow & (A \times B) \times C \\ \mathcal{R}.6 & A \to (B \to C) & \Rightarrow & (A \times B) \to C \\ \mathcal{R}.7 & A \to (B \times C) & \Rightarrow & (A \to B) \times (A \to C) \end{array} \tag{1.5}$$

Rules $\mathcal{R}.1$–$\mathcal{R}.4$, i.e., those rules handling the unit type, are confluent. Since the rules are trivially implemented in linear time, we assume here and henceforth that they are applied in a pre-processing stage, in which all occurrences of $\mathbf{1}$ are eliminated. [3]

After this stage, rules $\mathcal{R}.6$ and $\mathcal{R}.7$ are repeatedly applied as long as it is possible to do so. (We assume that types use a standard expression-tree representation in memory, and that each rule application is implemented as a transformation of this data structure.) Note that these rules can always simplify the structure of the right operand of $\to$, unless it is a primitive-type. The input is therefore brought to the normal form described by the following grammar

$$\tau ::= x \quad | \quad \tau \to x \quad | \quad \tau \times \tau \tag{1.6}$$

Next, rule $\mathcal{R}.5$ is repeatedly applied to "flatten" product-types. Now we can write every product ρ as

$$\rho = \left(\cdots \left((\tau_1 \times \tau_2) \times \tau_3 \right) \cdots \times \tau_k \right),$$

where τ_i, $i = 1, \ldots, k$, is either a primitive- or function-type. We will refer to $\tau_1, \ldots, \tau_k$ as the *terms* of ρ, and we will sometime even write ρ as $\prod_{i=1}^{k} \tau_i$ to emphasize the associative-commutative nature of product-types.

An algorithm for deciding first order isomorphism is to recursively compare the resulting normal forms: two nodes are isomorphic if they are of the same kind (product or function) and their operands are isomorphic. In function-nodes the comparison of arguments is straightforward: the left (right) operand of one node must be isomorphic to the left (right) operand of the other. In comparing product-nodes however we must solve an instance of the product polymorphism problem to check whether the terms of one node is pair-wise isomorphic to some permutation of the terms of the other node. If this comparison is not done carefully it adds to the complexity of the problem.

An even more serious inefficiency factor is that the system (1.5) (specifically, the distributive rule $\mathcal{R}.7$) may introduce an exponential blowup in the size of the representation. Rules $\mathcal{R}.1$–$\mathcal{R}.6$ do not increase the representation size. However, each application of $\mathcal{R}.7$ creates a duplicate copy of the subtree whose root is A. Repeated applications may produce a very large normal form representation. In the sequence of types $\{X_i\}$, defined by $X_0 = \mathsf{a}$ and $X_i = (\mathsf{b}_i \mathsf{c}_i)^{X_{i-1}}$ for $i > 0$, we have that $X_n \Rightarrow \mathsf{b}_n^{X_{n-1}} \mathsf{c}_i^{X_{i-1}}$ and successive applications of this rule to each occurrence of X_i, $i = n-1, \ldots, 1$, will lead to exponentially many copies of a in the normal form of X_n.

If graphs, rather than trees, are used to represent types, then an application of $\mathcal{R}.7$, can be implemented by *sharing* the node representing A. This sharing can be thought of as an application of a slightly different transformation

$$A \to (B \times C) \Rightarrow \begin{cases} (\alpha \to B) \times (\alpha \to C) \\ \alpha = A \end{cases}, \tag{1.7}$$

where a newly introduced symbolic variable α is represented as a pointer to the data-structure representation of type A.

Rittri [16] observed that using (1.7) ensures a polynomially sized representation of the normal form: Each application of transformation (1.7) adds one edge to the graph. The application reduces the nesting level of the $\times$ node, and this nesting level cannot be increased by the other rules. We obtain that the space of the graph normal form is $O(n^2)$ by noticing that initially there are at most n product-nodes, and that even though additional product-nodes may be created by $\mathcal{R}.6$, these nodes cannot take part in the other two rules.

To see that the representation can indeed by quadratic, consider the following example (written using the arithmetical notation):

$$\left(\mathsf{b}_1 \left(\mathsf{b}_2 \cdots \left(\mathsf{b}_{n-2} (\mathsf{b}_{n-1} \mathsf{b}_n^{\mathsf{a}_n})^{\mathsf{a}_{n-1}} \right)^{\mathsf{a}_{n-2}} \cdots \right)^{\mathsf{a}_2} \right)^{\mathsf{a}_1}, \tag{1.8}$$

whose normal form is

$$\mathsf{b}_1^{\mathsf{a}_1} \mathsf{b}_2^{\mathsf{a}_2 \mathsf{a}_1} \cdots \mathsf{b}_{n-1}^{\mathsf{a}_{n-1} \cdots \mathsf{a}_1} \mathsf{b}_n^{\mathsf{a}_n \cdots \mathsf{a}_1}. \tag{1.9}$$

This normal form consumes quadratic space if derived by applying $\mathcal{R}.7$ starting at the inner most parenthesis.

REMARK 1.4. *Deriving* (1.8) *starting at the outer most parenthesis, yields a linear space representation*

$$\mathsf{b}_1^{\alpha_1} \cdots \mathsf{b}_n^{\alpha_n},$$

where $\alpha_1 = \mathsf{a}_1$, *and* $\alpha_i = \mathsf{a}_i \alpha_{i-1}$ *for* $i = 2, \ldots, n$.

Having bound the space explosion, Rittri stopped short of giving a polynomial time algorithm for the problem. By noticing that the graph representation is acyclic, and by using a variant of Rittri's normal form, Considine [7] was able to reduce the runtime to polynomial. His algorithm partitions all nodes in the DAG representation of the input types into equivalence classes, such that all nodes in the same equivalence class are isomorphic. This partitioning is built in a bottom-up traversal of the DAGs, while maintaining a hash table mapping each node into the unique identifier of its equivalence class. The most difficult task was to determine whether product-nodes[4] are isomorphic. Two key properties made the $O(n^2 \log n)$ time and $O(n)$ space result possible:

1. *Expansion of product-types.* Considine showed that his normal form, which includes complete expansion of product-types, is such that each product consists no more than n terms.

[3] In the degenerate case in which one or both of the inputs is reduced to $\mathbf{1}$, the input types are isomorphic if and only if they both reduce to $\mathbf{1}$.

[4] We should note that Considine rules were different than Rittri's in that rule $\mathcal{R}.6$ was applied in the opposite direction. The resulting normal form is such that instead of A^{BCD}, it uses the equivalent representation $\left(\left(A^B \right)^C \right)^D$. Thus, strictly speaking, his normal form did not use product-nodes, other than in the upper most level. However, the alternative representation must still deal with the difficulties of associativity and commutativity as in the more familiar representation of products.

2. *Sorting product terms.* Since the graph is acyclic, terms in product-types must have been visited and classified by the bottom up traversal before the product itself. Each product-node is first normalized by sorting the identifiers of the equivalence classes of their terms. The fact that the order of terms is completely determined by this sorting makes it possible to employ a *hash-consing* technique to produce a unique identifier for each product-type, whereby partitioning product-type nodes into equivalence classes.

Our algorithm uses the same bottom-up classification of nodes into equivalence classes. However, the reduction of space to $O(n)$ and the time to $O(n \log^2 n)$ are made possible breaking away from the above principles. Specifically, the new algorithm is characterized by:

1. *Application of $\mathcal{R}.7$ to "outer-most" functions first.* As demonstrated in Remark 1.4 the space is kept linear if the distributive rule is applied starting at the outer most parenthesis.

2. *Unexpanded product-types.* The expansion of product-types leads to quadratic time and space. Instead, we describe a graph based representation, which keeps the space linear, and show that unexpanded products can still be efficiently compared.

3. *Unsorted product terms.* Isomorphism of product-nodes is decided by a procedure which can be thought of as hashing or range compaction, rather than sorting. A similar procedure is used to partition the multi-sets of products in each stage of the traversal into their equivalence classes.

Outline In Sec. 2 we develop a tool box for various multi-set partitioning problems, which we apply later for classifying unsorted product terms. The first such application is in Sec. 3 which describes our algorithms for the linear and product isomorphisms. Sec. 4 describes the normal form which we use for type comparisons, and shows that this normal form can be efficiently generated in a linear sized encoding, which we call the $\mathbf{P}/\mathbf{F}$-graph. Moreover, we show that the unexpanded products in the $\mathbf{P}/\mathbf{F}$-graph form a tree structure, such that each product inherits the terms of its parent.

Sec. 5 uses the multi-set partitioning toolbox in a procedure for comparing unexpanded products in this tree structure. Sec. 6 finetunes this procedure to its application in a bottom-up classification of the nodes of the $\mathbf{P}/\mathbf{F}$-graph. Finally, we present our main algorithm for deciding first order isomorphisms of simple types in Sec. 7. Sec. 8 lists some open questions.

2. Multi-set Partitioning Algorithms

For the purpose of processing product-nodes in which the terms are unsorted, we need a linear time procedure for comparing multi-sets. More generally, we develop in this section an algorithm for partitioning a collection of multi-sets of integers into equivalence classes. This algorithm runs in $O(n)$ time, where n is the size of the input representation, while using temporary (uninitialized) storage whose size is the maximal input value.

DEFINITION 2.1 (COMPACT INTEGER PARTITIONING). *Given integers $a_1, \ldots, a_n$, where $a_i \in [1, n]$ for $i = 1, \ldots, n$, the compact integer partitioning problem is to partition the input into its equivalence classes, i.e., all equal integers will be in the same partition (and only them).*

The output partitioning is presented with respect to the input: Each equivalence class is produced as a list of indices, $i_1, \ldots, i_m$, such that $a_{i_1} = \cdots = a_{i_m}$.

LEMMA 2.2. *Compact integer partitioning can be solved in $O(n)$ time and $O(n)$ space.*

PROOF. A standard bucket sort algorithm using n buckets achieves these bounds. □

More general than compact integer partitioning is the case that the input range is not restricted to the range $[1, n]$.

DEFINITION 2.3 (BROAD INTEGER PARTITIONING). *Given integers $a_1, \ldots, a_n$, where $a_i \in [1, U]$ for $i = 1, \ldots, n$, the broad integer partitioning problem is to partition the input into its equivalence classes.*

To deal with this problem, we first reduce the input range.

DEFINITION 2.4 (RENAMING). *Let $\mathcal{U}$ be an arbitrary domain and let $\Gamma \subseteq \mathcal{U}$, $|\Gamma| = n$. Then a partial function $\Omega : \mathcal{U} \to [1, n]$ is a* renaming *of Γ if Ω is defined on Γ and for any $a, b \in \Gamma$,*

$$a \neq b \Rightarrow \Omega(a) \neq \Omega(b).$$

Alg. 1 finds a renaming function for a sequence of integers drawn from the range $[1, U]$. The algorithm uses the standard trick of inverse pointers to maintain $O(1)$ access time into a sparse uninitialized array of arbitrary size. Note that main loop invariant: *After processing index i, then $\Omega[a_i] = t$ and $\mho[t] = a_i$, for some $t \in [1, \ell]$.*

Algorithm 1 `Rename`$(a_1, \ldots, a_n)$

Given the sequence $a_1, \ldots, a_n$, where $a_i \in [1, U]$, $i = 1, \ldots, n$, return **(i)** *$\ell = |\{a_1, \ldots, a_n\}|$ and* **(ii)** *a renaming function represented as an array $\Omega[1, \ldots, U]$, such that $\Omega[a_i]$ is a unique integer in the range $[1, \ell]$. The values of the other entries of Ω are arbitrary.*

```
 1: Ω ← new int[U] // An uninitialized array of size U
 2: ℧ ← new int[n] // The inverse mapping of Ω
 3: ℓ ← 0 // ℓ is the current number of distinct values in the input
 4: For i = 1, ..., n do // Compute Ω[a_i]
 5:    t ← Ω[a_i] // t may be arbitrary if the value of a_i is new
 6:    If 1 ≤ t ≤ ℓ andalso ℧[t] = a_i then
 7:       next i // No new mapping since a_i = a_j for some j < i
 8:    else // Create a new mapping entry
 9:       ℓ ← ℓ + 1 // A new distinct input value
10:       Ω[a_i] ← ℓ // Store the mapping entry
11:       ℧[ℓ] ← a_i // Record the inverse pointer
```

Renaming makes it possible to generalize Lemma 2.2.

LEMMA 2.5. *Broad integer partitioning can be solved in $O(n)$ time and $O(U + n)$ space.*

PROOF. After applying Alg. 1, we apply a *renaming process*, i.e., the replacement $a_i \leftarrow \Omega(a_i)$ for $i = 1, \ldots, n$. The problem is then reduced to compact integer partitioning. □

A more general partitioning problem is when the input consists of ordered pairs.

DEFINITION 2.6 (PAIR PARTITIONING). *Given a collection Γ of n pairs of integers $\langle a_1, b_1 \rangle, \ldots, \langle a_n, b_n \rangle$, where $a_i, b_i \in [1, U]$ for $i = 1, \ldots, n$, the pair partitioning problem is to partition Γ into its equivalence classes.*

LEMMA 2.7. *The pair partitioning problem can be solved in $O(n)$ time and $O(U+n)$ space.*

PROOF. Apply broad integer partitioning first on $a_1, \ldots, a_n$ to obtain an initial partitioning of Γ. Each of the resulting equivalence classes is then refined by broad integer partitioning with respect to the b_i's. □

Renaming with pair partitioning is also easy. Each pair is replaced by the index of its equivalence class. In fact, every partitioning algorithm gives rise to a corresponding renaming.

Lemma 2.7 can be generalized further.

LEMMA 2.8 (TUPLE PARTITIONING). *Given a collection Γ of n tuples of k integers each, where each integer is drawn from the range $[1, U]$, it is possible to partition Γ into its equivalence classes, in $O(nk)$ time and $O(U+n)$ extra space.*

PROOF. Similar to Lemma 2.7, however, instead of two passes we now have k passes. The input to the first pass is the entire collection Γ, and the output is a partitioning of Γ according to the first element of each tuple.

The output of pass i is a partitioning of Γ satisfying the following invariant: *all elements in the same partition have an equal i-prefix, i.e., the same first i integers in their tuples.* Pass i refines each partition by applying broad integer partitioning according to the i^{th} element of each tuple. Since broad integer partitioning is performed in linear time, the running time of a pass is linear in the sum of partition sizes, which is exactly $n = |\Gamma|$. Thus the total running time is $O(nk)$.

At the end of the k^{th} pass the tuple partitioning problem is solved. Broad integer partitioning requires (reusable) $O(U+n)$ space. In addition, only $O(n)$ space is required for storing the current partitioning of Γ in the form of indices to the input array. □

Notice that the time requirement in the above is linear in the size of the input, not the number of tuples. Also, observe that the algorithm for the tuple partitioning problem is in fact *incremental* in the sense that in the i^{th} pass we only examine the i^{th} integer in each tuple.

COROLLARY 2.9 (INCREMENTAL TUPLE PARTITIONING). *Let Γ be a collection of n tuples of k integers each, where each integer is drawn from the range $[1, U]$. Then, it is possible to incrementally partition Γ in k passes where the i^{th} component of each tuple is specified in the i^{th} pass, in $O(n)$ time for each pass and $O(U+n)$ extra space.*

A more challenging situation occurs in the case that the input consists of unordered tuples, rather than tuples. Next we will show that multi-set partitioning can also be solved in time linear in the size of the input.

DEFINITION 2.10 (MULTI-SET PARTITIONING). *Given a collection Γ of multi-sets of integers drawn from the range $[1, U]$, the multi-set partitioning problem is to partition Γ into its equivalence classes.*

LEMMA 2.11. *Multi-set partitioning can be solved in $O(n)$ time and $O(U+n)$ space, where n is the sum of sizes of all multi-sets.*

PROOF. First, Alg. 1 is invoked to rename all integers in the input to fit the range $[1, n]$. Then each multi-set is sorted using bucket-sort. We stress that we do not perform sorting of the initial multi-sets; we sort the renamed integers. (This sorting is possible in linear time since the renaming process is not order preserving.)

Next, the ordered multi-sets are partitioned according to size. Each such partition is a collection of ordered multi-sets of equal size; in other words, each partition is a collection of tuples of equal size. All that is left is to solve the tuple partitioning problem, employing Lemma 2.8 in each partition. □

3. Linear and Product Isomorphisms

In this section we use the multi-set partitioning algorithm in developing algorithms for linear and product isomorphisms. We first demonstrate how to decide product isomorphism in $O(n)$ time and space. Then we show a linear time and space reduction of linear isomorphism to product isomorphism.

After applying the unit-type pre-processing stage, product isomorphism has only two remaining axioms:

$$\begin{aligned} A \times B &= B \times A && \text{(Commutative)} \\ A \times (B \times C) &= (A \times B) \times C && \text{(Associative)} \end{aligned}$$

Commutativity and associativity allow product components to be reordered until the two types match. We see that product isomorphism is in essence a series of multi-set partitioning problems.

We use a *flattened products* normal form defined by the following grammar.

$$\begin{aligned} \rho &::= \prod_{i=1}^{m} \sigma && (m \geq 0) \\ \rho' &::= \prod_{i=1}^{m} \sigma && (m \geq 1) \\ \sigma &::= \rho \to x \quad | \quad \rho' \to \rho' \end{aligned} \tag{3.1}$$

The start symbol ρ denotes products of any number of terms, including 0 and 1, whereas ρ' denotes products of at least one term. Each term σ is a function-type. The symbol x denotes primitive-types.

The flattened products normal form requires that the terms of products are functions. This requirement can be obtained by applying the following simple transformation: if a primitive-type x is involved in a product, it must be rewritten as a function from a unit type (i.e., empty product) to x.

Conversely, function-types must receive and return products; the only exception being that the return type of a function can be a primitive-type. Also note that if this return type is a product then it must include at least one term. In other words, functions returning unity are not allowed in this normal form. This, and all other terms in the return type, must in turn be function-types. We need the following two transformation rules: **(i)** if the type of the argument of a function-type is a primitive-type x, then x must be rewritten as the singleton product $(\mathbf{1} \to x)$, and **(ii)** if a function-type has an operand which is also a function-type, then these must be separated by a singleton product.

Consider for example the following type,

$$(\mathsf{a} \times \mathsf{b} \to \mathsf{c}) \to (\mathsf{a} \times \mathsf{b} \times \mathsf{c} \times (\mathsf{a} \to \mathsf{b} \times \mathsf{c})). \tag{3.2}$$

Fig. 3.1 shows the abstract syntax tree of type (3.2) in the normal form (3.1).

We see in the figure for example that $\mathsf{a} \times \mathsf{b} \to \mathsf{c}$ was rewritten as

$$(\mathbf{1} \to \mathsf{a}) \times (\mathbf{1} \to \mathsf{b}) \to \mathsf{c}.$$

The grammar (3.1) produces abstract syntax trees in which function and product-types occur alternately on the path from the root to any leaf. We can thus define a height for each tree node, so that all nodes of odd (even) height represent function (product) types.

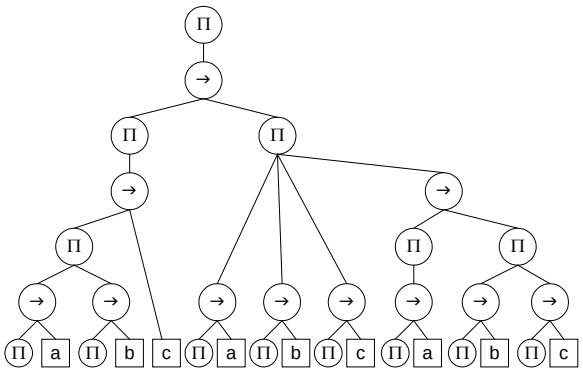

Figure 3.1: Type (3.2) after flattening

DEFINITION 3.1 (HEIGHT). *Let τ be a type written in the normal form* (3.1)*. Then, the height of a type, denoted $h(\tau)$, is the length of the longest path from τ to any leaf, i.e.,*

$$h(\tau) = \begin{cases} 0 & \text{if } \tau = x \text{ or } \tau \text{ is an empty product} \\ 1 + \max_{i=1}^{k} h(\sigma_i) & \text{if } \tau = \sigma_1 \times \cdots \times \sigma_k,\ k \geq 1 \\ 1 + \max(h(\rho), h(\tau)) & \text{if } \tau = \rho \to \tau \end{cases} \tag{3.3}$$

Edges in Fig. 3.1 were stretched so that nodes of the same height are drawn at the same level. Observe that product-types always have even heights and function-types always have odd heights. This can be easily proved by induction on the grammar of (3.1).

LEMMA 3.2. *If two types in the normal form* (3.1) *are isomorphic, they have the same height.*

PROOF. Omitted. □

THEOREM 3.3. *Product isomorphism can be decided in $O(n)$ time and space.*

PROOF. Consider the types represented by all of the nodes of the tree representations of flattened products normal forms of the two input types. We will label each of these $O(n)$ types with an identifier, such that two types are isomorphic iff they have the same identifier. These identifiers are integers drawn from the range $[1, cn]$ where c is some fixed constant.

Since two types cannot be equivalent unless their heights are the same, identifiers may be assigned in ascending order of heights. Let T_ι be the set of all types of height ι. The set T_0 is the set of primitive-types and empty products. The algorithm starts by passing T_0 to the broad integer partitioning algorithm. A renaming process then yields unique identifiers for all basic types, plus an identifier for the empty product.

The processing of T_ι, $\iota \geq 1$ depends on whether ι is even or odd. If ι is odd, then types in T_ι correspond to σ symbols in the grammar of the normal form, i.e., function-types. Equivalence among these are discovered using pair partitioning algorithm.

If however ι is even, then the types in T_ι are products, i.e., ρ symbols. We apply the multi-set partitioning algorithm to find all equivalence classes among these.

In both even and odd levels, we apply a renaming process that assigns identifiers to types in the current level, starting at the first unused identifier.

Each node is passed to a partitioning algorithm at most twice, first in the partitioning of nodes in its height, and then as component of its parent. Therefore the total input size in all invocations of partitioning algorithms is linear, and hence the total runtime of our algorithm is linear. □

The above algorithm is applicable also in the case that types use a DAG rather than a tree representation. The runtime in this case is linear in the number of nodes *and* the number of *edges* of the graph.

We now turn to the problem of linear isomorphism, which adds the currying axiom:

$$\begin{aligned} A \times B &= B \times A && \text{(Commutative)} \\ A \times (B \times C) &= (A \times B) \times C && \text{(Associative)} \\ (A \times B) \to C &= A \to (B \to C) && \text{(Currying)} \end{aligned}$$

The Currying axiom is no longer needed after exhaustively applying rule $\mathcal{R}.6$, [1]. Note that these applications do not increase the size of the representation. We can now bring the input to a form similar to the flattened products normal form (3.1). More precisely, the *linear normal form* is

$$\begin{aligned} \rho &::= \prod_{i=1}^{m} \sigma \qquad (m \geq 0) \\ \rho' &::= \prod_{i=1}^{m} \sigma \qquad (m \geq 1) \\ \rho'' &::= \prod_{i=1}^{m} \sigma \qquad (m \geq 2) \\ \sigma &::= \rho \to x \quad | \quad \rho' \to \rho'' \end{aligned} \tag{3.4}$$

which is almost identical to (3.1), except that ρ'' must have at least two terms. Once in this normal form, only the axioms used for product-types are necessary. We therefore have

THEOREM 3.4. *Linear isomorphisms can be decided in $O(n)$ time and space.*

4. The P/F-graph

To generalize the linear isomorphism algorithm to deal with the first order isomorphism problem, we must take care of distributivity, which, as noted above, may lead to an exponential blow-up of the representation. Inspired by the example set by Remark 1.4, we describe an algorithm for applying the distributive rule $\mathcal{R}.7$ more economically to produce a linear size representation in linear time.

We assume that the input was first brought to the form (3.4) in $O(n)$ time and space as described in the previous section. This input uses the standard expression tree representation. Repeated applications of $\mathcal{R}.7$ will then bring this input to the *distributive normal form* defined by the following grammar

$$\begin{aligned} \rho &::= \prod_{i=0}^{m} \sigma \qquad (m \geq 0) \\ \sigma &::= \rho \to x. \end{aligned} \tag{4.1}$$

The main difference between the two grammars is that the derivation

$$\sigma ::= \rho' \to \rho''$$

does not occur in (4.1). In other words, all functions must return a primitive-type. The transformation between (3.4) and (4.1) is carried out by invoking rule $\mathcal{R}.7$ whenever the return type of a function is not primitive.

To maintain a linear size representation, rule $\mathcal{R}.7$ is applied using (a multiple-terms version of) the distributive transformation (1.7): Consider an input node $A \to (B_1 \times \cdots \times B_k)$. Then, it follows from the grammar (3.4) that this node must in fact be of the form

$$\rho \to \big((\rho_1 \to \tau_1) \times \cdots \times (\rho_k \to \tau_k)\big), \tag{4.2}$$

where ρ and $\rho_1, \ldots, \rho_k$ are products, while $\tau_1, \ldots, \tau_k$ are either products- or primitive-types. Applying first the general version of (1.7) and then k times $\mathcal{R}.6$, we obtain

$$(\alpha \times \rho_1 \to \tau_1) \times \cdots \times (\alpha \times \rho_k \to \tau_k). \tag{4.3}$$

where the term α is represented as a pointer to the product ρ. We say that the product $\alpha \times \rho_i$ *inherits* all the terms of the product ρ.

Specifically, types in the distributive normal form (4.1) are represented as graphs (instead of trees), in which $\mathbf{P}$-nodes represent products. A $\mathbf{P}$-node v has a field $\varphi(v)$ storing the set of pointers to the terms of the product, which are all $\mathbf{F}$-nodes, i.e., nodes representing function-types. To represent term inheritance, v also has a field $\texttt{parent}(v)$ which is pointer to another $\mathbf{P}$-node, from which v inherits additional terms.

Let v be the $\mathbf{P}$-node of the product ρ in (4.3). Then, each product $\alpha \times \rho_i$ is represented by a $\mathbf{P}$-node v_i, such that $\varphi(v_i)$ stores the terms in the product ρ_i, while the shared term α is represented by making the assignment

$$\texttt{parent}(v_i) \leftarrow v.$$

An $\mathbf{F}$-node u has two main fields: **(i)** $\texttt{arg}(u)$, which is a pointer to the $\mathbf{P}$-node storing the function argument type, and **(ii)** $\texttt{ret}(u)$, which is a primitive-type specifying the function return type.

Although the representation breaks away from the standard expression tree representation, it is not arbitrarily general, in that sharing can only occur across `parent` edges. We require that the these edges make a tree $\mathcal{T}$, rooted at a dummy $\mathbf{P}$-node, denoted $\mathbf{P}_\perp$. $\mathbf{P}$-nodes are therefore initialized with their `parent` field pointing at $\mathbf{P}_\perp$. Node $\mathbf{P}_\perp$ has no terms, i.e., $\varphi(\mathbf{P}_\perp) = \emptyset$.

DEFINITION 4.1. *A $\mathbf{P}/\mathbf{F}$-graph is a rooted acyclic graph whose nodes are either* $\mathbf{P}$-nodes *or* $\mathbf{F}$-nodes.

Alg. 2 and Alg. 3 present two mutually recursive routines *`NormalizeProduct`* and *`FunctionIntoProduct`*. Together, the two describe a single pass traversal of an abstract syntax tree of the input grammar (3.4). The output is a $\mathbf{P}/\mathbf{F}$-graph of an isomorphic type in the distributive normal form.

Algorithm 2 *`NormalizeProduct`* (ρ)

Given an abstract syntax tree of a type ρ conforming to the grammar (3.4)*, return a* $\mathbf{P}$-node v *of an isomorphic type in the grammar* (4.1).

1: $v \leftarrow$ **new** $\mathbf{P}$-node // *Initially* $\texttt{parent}(v) = \mathbf{P}_\perp$, $\varphi(v) = \emptyset$
2: Let k and σ_i, $i = 1, \ldots, k$, be such that $\rho = \prod_{i=1}^{k} \sigma_i$
3: **For** $i = 1, \ldots, k$ **do** // *Normalized all terms in the product*
4: Let ρ_i and τ_i be such that $\sigma_i = \rho_i \to \tau_i$
5: $v_i \leftarrow$ *`NormalizeProduct`*(ρ_i)
6: $u \leftarrow$ *`FunctionIntoProduct`*(v_i, τ_i)
7: $\varphi(v) \leftarrow \varphi(v) \cup \varphi(u)$ // *Collect terms of u*
8: **Return** v

The recursive process relies on the fact that products and function-types occur alternately on a path from the root to any leaf. Accordingly, *`NormalizeProduct`* breaks the input product into its terms (line 2), which must all be function-types (line 4). The recursive conversion of the function type $\rho_i \leftarrow \tau_i$ is done in two steps, which must occur in order: At first (line 5), the function argument type is normalized, and the resulting $\mathbf{P}$-node product-node is stored in v_i. Only then (line 6), the function $\rho_i \leftarrow \tau_i$ can be brought into its normal form, which is a product of terms, where each such term may internally include pointers to the $\mathbf{P}$-node v_i.

Algorithm 3 *`FunctionIntoProduct`* (u, τ)

Given a $\mathbf{P}$-node u *and an abstract syntax tree of a type τ (which might be either a product- or a primitive-type) return v, a new* $\mathbf{P}$-node *describing a type isomorphic to the function-type $\rho \to \tau$, where ρ is the type represented by the* $\mathbf{P}$-node u.

1: $v \leftarrow$ **new** $\mathbf{P}$-node // *Initially* $\texttt{parent}(v) = \mathbf{P}_\perp$, $\varphi(v) = \emptyset$
2: **If** τ is a primitive-type x **then**
3: $w \leftarrow$ **new** $\mathbf{F}$-node
4: $\texttt{arg}(w) \leftarrow u$; $\texttt{ret}(w) \leftarrow x$ // *w represents the type $\rho \to x$*
5: $\varphi(v) \leftarrow \{w\}$
6: **Return** v
7: Let k and σ_i, $i = 1, \ldots, k$, be such that $\tau = \prod_{i=1}^{k} \sigma_i$
8: **For** $i = 1, \ldots, k$ **do** // *Process all terms in the return type*
9: Let ρ_i and τ_i be such that $\sigma_i = \rho_i \to \tau_i$
// $\rho \to \tau$ *is* $\rho \to ((\rho_1 \to \tau_1) \times \cdots \times (\rho_k \to \tau_k))$
10: $v_i \leftarrow$ *`NormalizeProduct`*(ρ_i)
11: $\texttt{parent}(v_i) \leftarrow u$ // *Share the common argument ρ*
12: $w \leftarrow$ *`FunctionIntoProduct`*(v_i, τ_i)
13: $\varphi(v) \leftarrow \varphi(v) \cup \varphi(w)$
14: **Return** v

Alg. 3 recursively traverses a function-type, returning an isomorphic $\mathbf{P}$-node. Lines 2–6 terminate the recursion in the case that the function return type is primitive. In this case, a singleton, non-inheriting product is returned. The remainder of this algorithm (lines 7–14) puts the transition between (4.2) and (4.3) in algorithmic terms.

Observe again that the recursive call in line 12 is done only after the function argument type was processed.

It is mundane to check that these two algorithms run in linear time and produce a linear sized representation.

Fig. 4.1 shows the result of applying these algorithm on the tree of Fig. 3.1. In the process, the two function-types whose right operand was a product-type were eliminated.

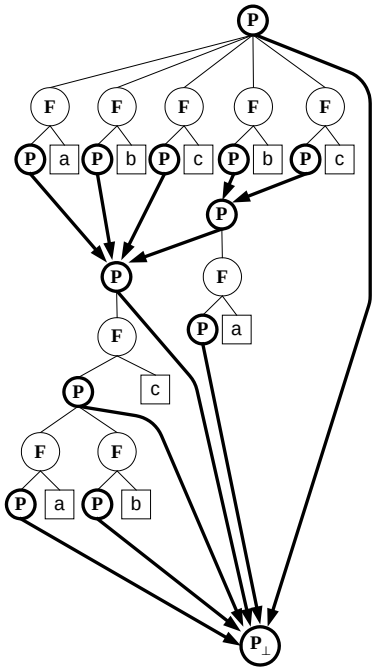

Figure 4.1: The $\mathbf{P}/\mathbf{F}$-graph of type (3.2)

The edges of the product tree are depicted in bold in the figure.

We see in the figure that these edges indeed do not form a cycle. It is easy to show that this is no coincidence by proving inductively that calls to `FunctionIntoProduct` create edges which always lead from the second to the first argument of this function.

DEFINITION 4.2 (EXPANDED TERMS). *The expanded terms of a* $\mathbf{P}$-node v, *denoted* $\phi(v)$, *are the union of terms of its ancestors in the product tree, i.e.,*

$$\phi(v) = \varphi(v) \cup \phi(\texttt{parent}(v)),$$

where $\phi(\mathbf{P}_\perp) = \emptyset$.

The following lemma shows that first order isomorphism of two types can be decided by bringing each of these types into their $\mathbf{P}/\mathbf{F}$ representation, and then traversing the two graphs in tandem, comparing at each stage the expanded terms of the current nodes.

LEMMA 4.3. *Two nodes* u, v *in a* $\mathbf{P}/\mathbf{F}$*-graph represent isomorphic types if and only if one of the following holds:*

1. u *and* v *are both* $\mathbf{F}$-nodes, $\texttt{ret}(u) = \texttt{ret}(v)$ *and* $\texttt{arg}(u)$ *and* $\texttt{arg}(v)$ *(recursively) represent isomorphic types.*
2. u *and* v *are both* $\mathbf{P}$-nodes, *and* $\phi(u) = \phi(v) = \emptyset$
3. u *and* v *are both* $\mathbf{P}$-nodes, *and there exists a bijection* π *from* $\phi(u)$ *to* $\phi(v)$, *such that every* $v' \in \phi(u)$ *(recursively) represents a type isomorphic to* $\pi(v')$.

PROOF. Omitted. □

If the terms in $\mathbf{P}$-nodes are expanded, then the size of the representation may increase to $O(n^2)$ (as in (1.9)). With this expansion, the problem becomes an instance of product isomorphisms, which, as explained in the previous section, can be solved in linear time. We can thus obtain a simple $O(n^2)$ time and space algorithm for the first order isomorphism problem, whereby improving upon the $O(n^2 \log n)$ best previous result.

To obtain a more efficient algorithm, we develop in the next section the machinery for comparing unexpanded products.

5. Tree Partitioning

In this section we further develop our partitioning algorithms to deal with the non-expanded representation of products in the tree of $\mathbf{P}$-nodes rooted at $\mathbf{P}_\perp$. Formally, out interest lies with a variant of the multi-set partitioning problem, in which the multi-sets are organized in an inheritance tree. Consider a tree $\mathcal{T}$ of n nodes such that a multi-set $\varphi(v)$ of integers is associated with each node $v \in \mathcal{T}$. The *expanded multi-set* of a node v is the union of multi-sets of the ancestors of v, i.e.,

$$\phi(v) \equiv \bigcup_{u \preceq v} \varphi(u).$$

These expanded multi-sets will be in our applications the expanded terms (Def. 4.2) of $\mathbf{P}$-nodes.

DEFINITION 5.1 (TREE PARTITIONING). *The* tree partitioning problem *is to solve the multi-set partitioning problem for the expanded multi-sets* $\{\phi(v) \mid v \in \mathcal{T}\}$.

A solution for the tree partitioning problem is useful in our attempt to partition the $\mathbf{P}$-nodes into equivalence classes, such that two $\mathbf{P}$-nodes are in the same class if and only if they represent isomorphic types. Types are isomorphic precisely when the expanded terms of the respective nodes are the same.

Let M denote the total number of elements in multi-sets of this tree, i.e.,

$$M \equiv \sum_{v \in \mathcal{T}} |\varphi(v)|.$$

We can assume that the integers in the input to the problem are condensed so that

$$\bigcup_{v \in \mathcal{T}} \varphi(v) = [1, m].$$

(This condition can be ensured by a simple application of a renaming process.)

Fig. 5.1a shows an example of a tree with $n = 8$ nodes with their associated multi-sets (only four of which are non-empty). In the example, $m = 4$ distinct integers take part in these multi-sets. The total number of elements in these multi-sets is $M = 9$.

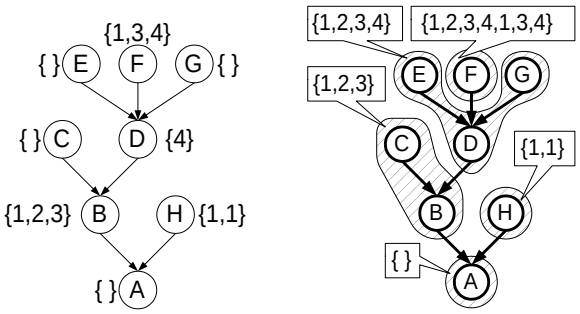

Figure 5.1: A small multi-set tree (a) and its tree partitioning (b)

We have for nodes E and F, for instance,

$$\begin{aligned}
\varphi(\mathsf{E}) &= \emptyset \\
\varphi(\mathsf{F}) &= \{1, 3, 4\} \\
\phi(\mathsf{E}) &= \{1, 2, 3, 4\} \\
\phi(\mathsf{F}) &= \{1, 2, 3, 4, 1, 3, 4\}
\end{aligned}$$

Fig. 5.1b depicts the solution of the tree partitioning problem for the multi-set tree of Fig. 5.1a. We see that there are 5 partitions:

$$\{\mathsf{A}\}, \{\mathsf{H}\}, \{\mathsf{B}, \mathsf{C}\}, \{\mathsf{D}, \mathsf{E}, \mathsf{G}\}, \{\mathsf{F}\}. \tag{5.1}$$

The callout attached to each partition shows the expanded multi-set of all nodes in this partition. For example, $\{1, 2, 3, 4\}$ is the expanded multi-set of the partition $\{\mathsf{D}, \mathsf{E}, \mathsf{G}\}$.

The naive solution to the tree partitioning problem is by directly computing the expanded multi-sets $\phi(v)$. To do so, represent an expanded multi-set $\phi(v)$ as an integer array $\mathbf{Count}_v[1, \ldots, m]$ such that $\mathbf{Count}_v[i] = k$ if integer i occurs k times in $\phi(v)$. Array $\mathbf{Count}_v$ can be easily computed from $\varphi(v)$ and $\mathbf{Count}_u$, where u is v's parent.

After having obtained the arrays $\mathbf{Count}_v$, the tree partitioning problem becomes the partitioning problem of these arrays, viewed as m-sized tuples.

The total time for computing the expanded multi-sets in the above fashion is $O(nm + M)$. We now present a more efficient algorithm which relies on the *dual* representation in which, instead of associating a multi-set of integers with each node, a multi-set of nodes is associated with each integer. The performance gain is due to the fact that the multi-set of nodes in which a value participates is most

often a subtree of $\mathcal{T}$. With an implicit ordering of the tree nodes obtained by π, a pre-order traversal of the tree, subtrees can be simply encoded as intervals.

In our example, let the pre-order traversal be

$$\pi = (\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}, \mathsf{F}, \mathsf{G}, \mathsf{H}).$$

We see that the descendants of any given node form an interval. For instance,

$$\begin{aligned} \text{descendants}(\mathsf{D}) &= \{\mathsf{D}, \mathsf{E}, \mathsf{F}, \mathsf{G}\}, \\ \text{descendants}(\mathsf{B}) &= \{\mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}, \mathsf{F}, \mathsf{G}\}. \end{aligned}$$

A *family* F_i, $i = 1, \ldots, m$, is the multi-set of nodes whose multi-sets contain i. More precisely, if i occurs k times in $\varphi(v)$, then v occurs k times in F_i. In our example, four such families are defined:

$$\begin{aligned} F_1 &= \{\mathsf{B}, \mathsf{F}, \mathsf{H}, \mathsf{H}\} \\ F_2 &= \{\mathsf{B}\} \\ F_3 &= \{\mathsf{B}, \mathsf{F}\} \\ F_4 &= \{\mathsf{D}, \mathsf{F}\} \end{aligned} \tag{5.2}$$

Each family F_i defines at most $|F_i|$ distinct intervals in π, one for each distinct node in F_i. These intervals partition π into at most $2|F_i| + 1$ segments. This partitioning uniquely defines the number of occurrences of i in each of the tree nodes: Consider any arbitrary such segment, and let v range over the nodes of this segment. Then, the multiplicity of the value i in $\phi(v)$ is the same.

Instead of explicitly writing the multiplicity of i in $\phi(v)$ for every such node v, we associate the multiplicity with the entire segment. We shall even refer to this multiplicity as the *segment id*. A partitioning of the pre-order traversal into segments and their identifiers is called a *segment partitioning*. The segment partitioning of a family F is denoted ∇F. Let $\mathrm{id}_i(v)$ be the id of the segment of a node v in ∇F_i. Then, the essence of the dual representation is that

$$\mathrm{id}_i(v) = \mathbf{Count}_v[i]. \tag{5.3}$$

We therefore have

LEMMA 5.2. *The sequence* $\mathrm{id}_1(v), \ldots, id_m(v)$ *uniquely determines* $\phi(v)$.

Fig. 5.2 depicts the segment partitionings of the families of (5.2).

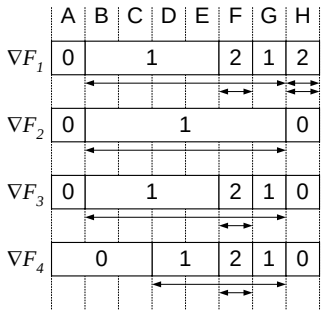

Figure 5.2: The segment partitionings of the families of Fig. 5.1a

We see in the figure that F_3 (for example) defines two intervals which induce a partitioning of π into four segments. The largest of these segments has nodes $\mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}$. The id of this segment is 1, and indeed we can see in Fig. 5.1b the value 3 occurs exactly the same number of times (i.e., 1) in the expanded multi-sets of each of these nodes. The id 1 is also given to the singleton segment of G; this is because the value 3 occurs once also in the expanded multi-set of G

Also, note in the figure that $\phi(v)$ can be read by inspecting the segment id's in each of the partitionings. For example, reading the first row of the column headed F we see that 1 occurs twice in $\phi(\mathsf{F})$.

The *intersection* of two segment partitionings P_1 and P_2 is also a segment partitioning, written as $P_1 \times P_2$. It can be obtained by intersecting each of the segments of P_1 with each of the segment of P_2. The identifiers of the segments in $P_1 \times P_2$ are the *renamed* pairs of identifiers of the originating segments from P_1 and P_2. We can safely reuse the old identifiers, i.e., the new identifiers are allocated starting from 0.

Fig. 5.3 depicts the intersection of the segment partitionings of ∇F_1 and ∇F_2 from (5.2).

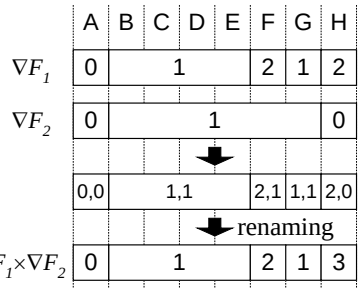

Figure 5.3: Computing the intersection of the two segment partitionings ∇F_1 and ∇F_2 defined by Fig. 5.1a.

The third row in the figure shows the intermediate stage in which the segments in the intersection still use pairs of integers as identifiers. For example, $\langle 1, 1\rangle$ is the identifier of the segment containing nodes B, C, D, and E. This identifier was renamed to 1. Note that the other segment (singleton with G) with the pair identifier $\langle 1, 1\rangle$ was also renamed to 1.

We represent segment partitionings as a sorted array of the segment end-points with their associated identifiers. The intersection of two such partitionings whose sizes are s_1 and s_2 is carried out by merging their arrays in $O(s_1 + s_2)$ time into a single $s_1 + s_2$ sized array. Lemma 2.7 can then be employed to rename the pair-identifiers in the merged array.

It follows from Lemma 5.2 that the tree partitioning problem is solved by computing

$$\nabla F_1 \times \cdots \times \nabla F_m.$$

LEMMA 5.3. *There is an* $O(n + M \log m)$ *time and* $O(n + M)$ *space algorithm solving the tree partitioning problem.*

PROOF. To compute $\nabla F_1 \times \cdots \times \nabla F_m$ we build a balanced binary tree whose leaves are $\nabla F_1, \ldots, \nabla F_m$. In each internal node we compute the intersection of the two segment partitionings of its two children. The segment partitioning at the root of this tree is exactly the tree partitioning.

Since all the partitionings propagate to the root, we have that the total size of all partitionings at each tree level, and thus the work to generate the next level is $O(M)$. Since the number of levels is logarithmic, we have that the total time for computing

$$\nabla F_1 \times \cdots \times \nabla F_m$$

is $O(M \log m)$. □

Fig. 5.4 depicts the balanced binary tree of the families of (5.2).

We see in the figure that the segment partitioning at the root of this binary tree, i.e., $\nabla F_1 \times \nabla F_2 \times \nabla F_3 \times \nabla F_4$, partitions the

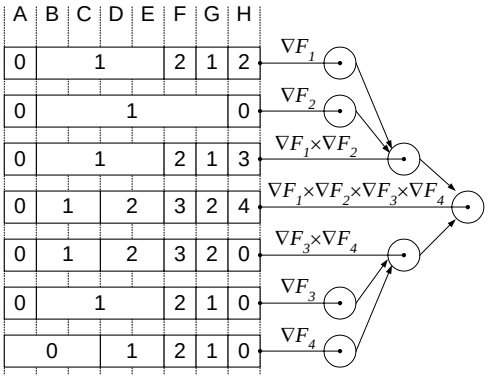

Figure 5.4: The balanced binary tree of the families of Fig. 5.1

ordering π into 6 segments. The segment of types D and E has $\mathrm{id} = 2$. This is also the id of the segment of G. Together, these two segments represent the partition $\{\mathsf{D}, \mathsf{E}, \mathsf{G}\}$. We have thus obtained the desired partitioning (5.1) of the tree in Fig. 5.1a.

6. Incremental Tree Partitioning

The tree partitioning problem (Def. 5.1) solved in the previous section does not capture in full the intricacies of the bottom up classification into isomorphism classes of the nodes of a $\mathbf{P}/\mathbf{F}$-graph. The difficulty is that the terms of $\mathbf{P}$-nodes in any given height are $\mathbf{F}$-nodes. These $\mathbf{F}$-nodes must be classified prior to the classification of the $\mathbf{P}$-nodes in this height. The algorithm behind Lemma 5.3 however assumes that all multi-sets members are directly comparable. It is applicable only in the case when all terms are basic types.

In this section, we develop the algorithm which after having classified all the $\mathbf{P}$-nodes up to height ι, will use this information to classify the $\mathbf{F}$-nodes in height $\iota + 1$. The identifier found in the classification of these $\mathbf{F}$-nodes must take part in the classification of the $\mathbf{P}$-nodes at height $\iota + 2$.

To this end, this section deals with a more general variant of the tree partitioning problem, in which the multi-sets are supplied in a piecemeal fashion. In this variant, the different possible values of the multi-sets in the tree nodes are exposed in iterations. The algorithm for this variant will add another logarithmic factor to the time complexity.

The requirements from a data structure for the *incremental tree partitioning problem* are best defined in terms of the dual representation. Given a tree $\mathcal{T}$, this data structure must support two kinds of operations, which might be interleaved: *insertions* of the sequence of families $F_1, F_2, \ldots$ and *classifications* of a sequence of node sets $T_1, T_2, \ldots$

In our application, the multi-set of nodes F_j is inserted after having discovered that a certain collection of $\mathbf{F}$-nodes belong in the isomorphism class whose identifier is j. (These identifiers are allocated consecutively.) The set of nodes T_j is classified when we carry out the classification of nodes whose height is $2j$.

Accordingly, the sequence $\{T_j\}$ cannot be arbitrary. We demand that these sets are disjoint, that $\bigcup_j T_j = \mathcal{T}$ and that the data structure is never required to classify a node before its parent. The interleaving of the two input sequences is also subject to restrictions. If a node was classified then it can never appear again in any insertion.

With these restrictions, we use the notation $\mathcal{T}.\texttt{classify}(T)$ and $\mathcal{T}.\texttt{insert}(F_j)$ to describe these operations. Our main objective is to minimize the resources for processing the entire interleaved sequence of data structure operations.

LEMMA 6.1. *Incremental tree partitioning can be solved in*

$$O(n + M \log m + n \log n \log m)$$

time and $O(n + M)$ space.

PROOF. We use a lazy representation of an infinite complete binary tree, similar to the binary tree of Lemma 5.3, The leaves of this tree are given by the infinite sequence $\nabla F_1, \nabla F_2, \ldots$

Fig. 6.1 shows (part of) this tree, after families $\nabla F_1, \ldots, \nabla F_7$ have been inserted.

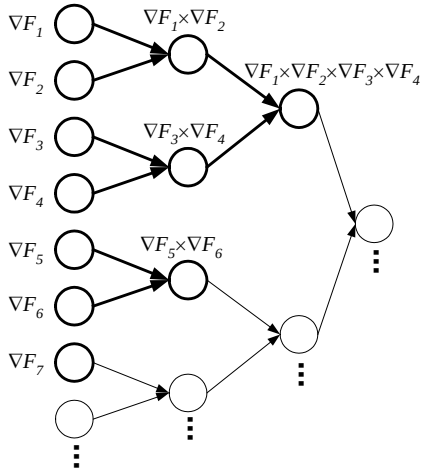

Figure 6.1: An embedding of seven families into an infinite balanced binary tree

This infinite tree is used to guide the computation of the intersection of the partitioning which were inserted so far: we delay the intersection of partitionings in an internal node until *both* its children exist. A *temporary root* is a node in which the partitioning was computed, but not in its parent.

In the figure the nodes at which partitionings were intersected are drawn with thicker lines. Specifically, at this stage we have computed $\nabla F_1 \times \nabla F_2$, $\nabla F_3 \times \nabla F_4$, $\nabla F_1 \times \nabla F_2 \times \nabla F_3 \times \nabla F_4$, and $\nabla F_5 \times \nabla F_6$. There are three temporary roots in figure, which are the nodes corresponding to $\nabla F_1 \times \nabla F_2 \times \nabla F_3 \times \nabla F_4$, $\nabla F_5 \times \nabla F_6$ and ∇F_7

Assume that a new family F_8 is inserted. We first calculate its segment partitioning ∇F_8, and proceed to compute the following three intersections:

$$\begin{aligned} P_1 &\equiv \nabla F_8 \times \nabla F_7, \\ P_2 &\equiv P_1 \times (\nabla F_5 \times \nabla F_6), \\ P_3 &\equiv P_2 \times (\nabla F_1 \times \cdots \times \nabla F_4). \end{aligned}$$

After this insertion we will have a single temporary root.

Note that after j families were inserted, there are at most $\lceil \log_2 j \rceil$ temporary roots. Also note that the total time for all the insertions is the same as in the non-incremental tree partitioning problem, i.e., $O(n + M \log m)$.

The classification of a set T is carried out by consulting the list $P_1, \ldots, P_r$ of the segment partitionings at the temporary roots. Recall that P_i is represented as a sorted array. Since the size of this array is bounded by n, we can support searches in P_i in $O(\log n)$ time. For each $v \in T$, we search for the id of the segment which contains v, in P_i for $i = 1, \ldots, r$. Since $r \leq \lceil \log_2 m \rceil$, the total time for these searches is $O(\log m \log n)$.

After obtaining an r-tuple of ids for all $v \in T$, we apply a tuple partitioning algorithm to classify T. In order to keep the space linear, we cannot actually store $|T|$ tuples of length r. Therefore, we

will use the *incremental tuple partitioning algorithm*. Specifically, we will use $|T|$ memory cells to find the first elements of the tuples, pass them to the tuple partitioning algorithm, and proceed to find the second elements of the tuples, etc. The total time for the `classify` operation is $O(|T| \log m \log n)$, while using $O(n + M)$ space.

Since every node $v \in \mathcal{T}$ can take part in a classification operation at most once, the total time for all classifications is

$$O(n \log n \log m).$$

The total time for insertion and classification operations is

$$O(n + M \log m + n \log n \log m),$$

while the total space used is $O(n + M)$. □

7. Algorithm for First Order Isomorphism

Having developed the algorithms for generating the linear size $\mathbf{P}/\mathbf{F}$ representation, and for efficiently comparing the multi-sets ϕ without actually creating them, we are ready to describe the main result of this paper: an efficient algorithm for deciding first order isomorphisms. In essence, the algorithm uses Lemma 4.3. A naive recursive application of the lemma may lead to an exponential running time. To bound the time complexity, we instead traverse the graphs bottom-up, classifying the nodes into their isomorphisms equivalence classes as we do so.

The bottom-up traversal is guided by height, where all nodes of the same height are processed together. Height is defined as in Def. 3.1. Alg. 4 shows how heights can be computed in linear time even in the non-expanded, $\mathbf{P}/\mathbf{F}$ representation.

Algorithm 4 `Height`(v)

Given a node v in a $\mathbf{P}/\mathbf{F}$-graph, ensure that $h(v')$ stores the height of v' for all nodes v' reachable from v and return $h(v)$.

```
 1: If v was visited then
 2:     Return h(v)
 3: mark v as visited
 4: If v is an F-node then
 5:     h(v) ← 1 + Height(arg(v)); return h(v)
 6: If v = P⊥ then // φ(v) = ∅ and parent(v) = nil
 7:     h(v) ← 0; return h(v) // Recursion base
    // v must be an ordinary P-node
 8: h(v) ← Height(parent(v))
 9: For all u ∈ φ(v) do // recurse on all (non-expanded) terms
10:     h(v) ← max(h(v), 1 + Height(u))
11: Return h(v)
```

Given a node v, the algorithm uses a standard recursive depth first search to visit, compute and store the height of every node v' reachable from v. Lines 4–5 deal with the case that v is an $\mathbf{F}$-node. The recursive call in this case is only on $\texttt{arg}(u)$, since $\texttt{ret}(v)$ must be a primitive-type.

Another easy case is that v is the $\mathbf{P}_\perp$. Since there are no terms in this product-node, its height is 0. Lines 8–11 deal with ordinary $\mathbf{P}$-nodes. The height of such nodes is one more than the maximum height of all expanded terms. The reason is that in line 8 we do not add 1 to $\texttt{Height}(\texttt{parent}(v))$ is that the expanded terms include the terms $\phi(\texttt{parent}(v))$, and not $\texttt{parent}(v)$ as a term.

Once the height of all nodes in $\mathbf{P}/\mathbf{F}$-graph is computed, Alg. 5 can be invoked to partition these nodes into equivalence classes. We assume that unique identifiers, drawn from the range $[1, n]$, are given to all basic types. To process non-basic types, the algorithm relies on the fact that nodes cannot represent isomorphic types unless they are of the same kind and the same height. Accordingly, the nodes of G are processed by height.

Algorithm 5 `NodesPartitioning` (G)

Given a $\mathbf{P}/\mathbf{F}$-graph G representing a type in the distributive normal form (4.1)*, return a partitioning Λ of all the nodes of G into equivalence classes, such that two nodes are in the same class if and only if they represent isomorphic types.*

```
 1: Let T be an incremental tree partitioning data-structure for the
    tree of P-nodes of G
 2: j ← 0 // The identifier of current isomorphism class
 3: Let r be the root of G
 4: l ← Height(r)
 5: For ι = 1, ..., l do // Process the nodes by height
 6:     Let T_ι ← {v ∈ G | h(v) = ι}
 7:     If ι is even then // T_ι is a collection of P-nodes
 8:         Λ ← Λ ∪ T.classify(T_ι)
 9:     else // T_ι is a collection of F-nodes
10:         Partition T_ι using pair partitioning
11:         Let the resulting partition be T_ι = C_1 ∪ ··· ∪ C_k
12:         Λ ← Λ ∪ {C_1, ..., C_k}
            // Update T
13:         For i = 1, ..., k do // Inserting a new family
14:             j ← j + 1 // Process a new isomorphism class j
15:             Let F_j be the multi-set of P-nodes with a term in C_i
16:             T.insert(F_j)
17: Return Λ
```

The main data-structure used by the algorithm is incremental tree partitioning (Lemma 6.1). Nodes at even height are $\mathbf{P}$-nodes. The classification of these nodes is carried out by querying this data-structure.

Lines 10–16 in the algorithm take care of $\mathbf{F}$-nodes. Classification of these nodes is carried out by a simple pair partitioning algorithm. We then generate identifiers for each of the isomorphism classes. All $\mathbf{F}$-nodes take parts as terms of $\mathbf{P}$-nodes. We must make sure that two $\mathbf{F}$-nodes in the same isomorphism class are regarded as equal when comparing $\mathbf{P}$-nodes in the next iteration. Line 15 defines the multi-set F_j of $\mathbf{P}$-nodes in which isomorphic $\mathbf{F}$-nodes are terms. Note that F_j is a multi-set since a $\mathbf{P}$-node may have several terms belonging to C_i. In line 16 the incremental tree partitioning data structure is updated.

LEMMA 7.1. *If G has n nodes and $O(n)$ edges then, Alg. 5 runs in $O(n \log^2 n)$ time and while consuming $O(n)$ space.*

PROOF. We first note that computing the height as in Alg. 4 requires linear time, since every node and every edge is visited at most once.

The algorithm uses linear space, since the two main procedures it invokes: incremental tree partitioning algorithm (lines 8 and 16 and pair partitioning (line 10) use linear space.

The running time of all the applications of the pair partitioning algorithm is $O(n)$ (see Lemma 2.7).

The total number of families inserted is $O(n)$. Moreover, the total size of those families is also $O(n)$, and all the sets of classified nodes are disjoint. It follows therefore from Lemma 6.1 that the total time of all the operations performed on $\mathcal{T}$ is $O(n \log^2 n)$. □

The bottom-up node classification of Alg. 5 can be used to solve the first order isomorphism problem. To do so, we first create the $\mathbf{P}/\mathbf{F}$-graphs of the two input types, and then merge these graphs,

by e.g., making their roots descendants of a new $\mathbf{P}$-node. (The $\mathbf{P}_\perp$ nodes of the respective graphs must be unified.) Alg. 5 is then invoked on the merged graph. The inputs are isomorphic if and only if these two roots are placed in the same equivalence class.

THEOREM 7.2. *First order isomorphism can be decided in*

$$O(n \log^2 n)$$

time and $O(n)$ space, where n is the size of the input.

PROOF. As noted above the $\mathbf{P}/\mathbf{F}$-graph representation uses linear space. Moreover, bringing the input to this representation requires linear time.

The complexity of comparing inputs in the $\mathbf{P}/\mathbf{F}$-graph representation is given by Lemma 7.1. □

8. Open Problems

Further research may take the following directions.

1. The only lower bound for the first order isomorphism problem is the trivial information theoretic linear time. It would be interesting to bridge this gap by either reducing the time complexity of our main algorithm even further, or obtaining better lower bounds.

2. Most programming language allow the user to define types indirectly, i.e., by giving names to non-primitive-types, and then using these names in the definition of more complex types. We would like therefore to generalize our algorithm to deal with input represented as a graph rather than the standard expression tree.

3. Perhaps the most important problem which this paper leaves open is efficient algorithms for *subtyping* (of products, functions, or both) which include the distributive and the currying axioms.

9. REFERENCES

[1] A. Andreev and S. Soloviev. A deciding algorithm for linear isomorphism of types with complexity $O(n \log^2(n))$. *Lecture Notes in Computer Science*, 1290:197ff, 1997.

[2] J. Auerbach, C. Barton, and M. Raghavachary. Type isomorphisms with recursive types. Technical Report RC 21247, IBM Research Division, Yorktown Heights, New York, August 1998.

[3] J. Auerbach and M. C. Chu-Carroll. The mockingbird system: A compiler-based approach to maximally interoperable distributed programming. Technical Report RC 20178, IBM Research Division, Yorktown Heights, New York, February 1997.

[4] D. A. Basin. Equality of terms containing associative-commutative functions and commutative binding operators is isomorphism complete. In *10th International Conference on Automated Deduction*, pages 251–260. Springer-Verlag New York, Inc., 1990.

[5] K. B. Bruce, R. D. Cosmo, and G. Longo. Provable isomorphisms of types. *Mathematical Structures in Computer Science*, 1:1–20, 1991.

[6] K. B. Bruce and G. Longo. Provable isomorphisms and domain equations in models of typed languages. In *Proc. of the 7th annual ACM symposium on Theory of computing*, pages 263–272. ACM Press, May 1985.

[7] J. Considine. Deciding isomorphisms of simple types in polynomial time. Technical report, CS Department, Boston University, April 2000.

[8] R. D. Cosmo. Type isomorphisms in a type-assignment framework. In *Proc. of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 200–210. ACM Press, 1992.

[9] R. Di Cosmo. *Isomorphisms of types: from λ-calculus to information retrieval and language design*. Birkhauser, 1995. ISBN-0-8176-3763-X.

[10] M. Fiore, R. D. Cosmo, and V. Balat. Remarks on isomorphisms in typed lambda calculi with empty and sum types. In *Proc. of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS'02)*, July 2002.

[11] J. Y. Gil. Subtyping arithmetical types. In *27th Symposium on Principles of Programming Languages, POPL'01*, pages 276–289, London, England, Jan.17–19 2001. ACM SIGPLAN — SIGACT, ACM Press.

[12] R. Gurevič. Equational theory of positive numbers with exponentiation. *American Mathmatical Society*, 94(1):135–141, May 1985.

[13] W. A. Howard. The formulaes-as-types notion of construction. In J. R. Hindley and J. P. Seldin, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*, pages 479–490. Academic Press, 1980.

[14] S. Jha, J. Palsberg, and T. Zhao. Efficient type matching. In *Proc. of the 5th Foundations of Software Science and Computation Structures*, Grenoble, France, April 2002.

[15] J. Palsberg and T. Zhao. Efficient and flexible matching of recursive types. In *Proc. of LICS'00, 15th Annual IEEE Symposium on Logic in Computer Science*, June 2000.

[16] M. Rittri. Using types as search keys in function libraries. *Journal of Functional Programming*, 1(1), 1989.

[17] M. Rittri. Retrieving library identifiers via equational matching of types. In *10th International Conference on Automated Deduction*, pages 603–617, 1990.

[18] S. V. Soloviev. The category of finite sets and cartesian closed categories. *Journal of Soviet Mathematics*, 22(3):1387–1400, 1983.

[19] A. Tarski. *A Decision Method for Elementary Algebra and Geometry*. University of California Press, Berkeley, CA, 2nd edition, 1951.