Under consideration for publication in Math. Struct. in Comp. Science

Efficient Algorithms for Isomorphisms of Simple Types[†]

JOSEPH (YOSSI) GIL[‡] and YOAV ZIBIN

Technion—Israel Institute of Technology

Email: `yogi` | `zyoav @ cs.technion.ac.il`

Received October 2003; Revised July 2004

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, thereby improving upon the $O(n \log n)$ time of the best previous algorithm for this problem.

1. Introduction

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

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

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 (Soloviev, 1983; Bruce et al., 1991) and intuitionistic logic (Howard, 1980).

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.[†] (A detailed treatise of this application can be found in Di Cosmo's book (1995), 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

[†] A preliminary version of this paper was published in the proceedings of POPL'03 (Zibin et al., 2003).
[‡] Research supported in part by the generous funding of the Israel Science Foundation, grant No. 128/02.
[†] Besides being sufficient for the proof of equations such as (1.1).

by the following *general grammar*

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

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

In defining the isomorphism relation we shall use the following seven axiom schemas.

$(\mathcal{A}.1)$	$A \times \mathbf{T} = A$	
$(\mathcal{A}.2)$	$A \to \mathbf{T} = \mathbf{T}$	
$(\mathcal{A}.3)$	$\mathbf{T} \to A = A$	
$(\mathcal{A}.4)$	$A \times B = B \times A$	(Commutative)
$(\mathcal{A}.5)$	$A \times (B \times C) = (A \times B) \times C$	(Associative)
$(\mathcal{A}.6)$	$(A \times B) \to C = A \to (B \to C)$	(Currying)
$(\mathcal{A}.7)$	$A \to (B \times C) = (A \to B) \times (A \to C)$	(Distributive)

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

For a long time, the problem of deciding first order isomorphisms of simple types was thought to require exponential time (Bruce et al., 1991). It was recently shown (Considine, 2000) 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 $\mathbf{T}$) was proved to be complete for the Cartesian closed categories (Bruce et al., 1991; Soloviev, 1983). Since the models of the lambda calculus with unit, product- and function-types are exactly the Cartesian closed categories (Bruce et al., 1991), the set is also complete for the type isomorphisms we examine. Through the Curry-Howard isomorphism (Howard, 1980), 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 (Rittri, 1990) 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. It was even argued (Barthe and Pons, 2001) that type isomorphisms can be employed in proof reuse.

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 (Basin, 1990; Di Cosmo, 1995) 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 (Di Cosmo, 1992) that second order isomorphisms are insufficient to describe these, although the addition of one more axiom suffices.

Recursive variants of the type isomorphism problem at our hand were also considered in the literature. In the Mockingbird project the recursive type system comprised of product- and function-types (Auerbach and Chu-Carroll, 1997; Auerbach et al., 1998; Palsberg and Zhao, 2000). Gil (2001) describes how algorithms for polynomial equality can be used for deciding isomorphism in the "algebraic type system", i.e., the recursive type system comprising of union- and product-types.

The more general isomorphism problem, for a non-recursive type system which includes product-, union- *and* function-types is equivalent to Tarski's *high school algebra problem* (Tarski, 1951). Such a system does not have a finite and complete set of axioms. Nonetheless, there exists a (non-polynomial) algorithm for determining isomorphism (Gurevič, 1985). There also exists a (non-polynomial) algorithm for deciding isomorphism in the recursive "algebraic type system" (Gil, 2001). Finally, we should mention that adding empty and sum types breaks down the relationship between the equational theory and type isomorphisms (Fiore et al., 2002).

1.2. *Definitions: The First Order Isomorphism Problem and its Variants*

In this paper, we concentrate on first order isomorphism and two restricted variants (product and linear isomorphism). We now make the necessary definitions in order to give a precise statement of the problem and its variants.

Next we define four successive theories of isomorphism of types.

Definition 1.1. Let Equality be the theory of equality of types defined as the set of propositions obtained by the deductive closure of the axiom schema

$$(\mathcal{A}.0) \qquad A = A \quad \text{(Reflexive)}$$

and the following four inference rules.

$$\frac{A = B}{B = A} \qquad \text{symmetry}$$

$$\frac{A = B, B = C}{A = C} \qquad \text{transitivity}$$

$$\frac{A = B, C = D}{A \times C = B \times D} \qquad \text{congruence of } \times$$

$$\frac{A = B, C = D}{A \to C = B \to D} \qquad \text{congruence of } \to$$

Thus, Equality is the usual theory of equality, sometimes denoted as Th^0 (Considine, 2000).

Definition 1.2. Let $\mathsf{Product}$ be the theory $\mathsf{Equality}$ augmented with axiom schemas $\mathcal{A}.1$–$\mathcal{A}.5$. Let Linear be the theory $\mathsf{Product}$ augmented with axiom schema $\mathcal{A}.6$. Let First be the theory Linear augmented with axiom schema $\mathcal{A}.7$.

Theory $\mathsf{Product}$ adds the unit axioms to the theory of equality as well as the rules of commutative and associative products. The currying axiom is added in theory Linear. Finally, First is the theory of first order isomorphisms, which is often referred to in the literature as $Th^1_{\times T}$ (Bruce and Longo, 1985; Bruce et al., 1991; Di Cosmo, 1995).

When $\mathbf{T}$ does not occur in the input, it is convenient to use theory variants which do not include the unit axioms.

Definition 1.3. Let $\mathsf{Product}^-$ be the theory $\mathsf{Equality}$ augmented with axiom schemas $\mathcal{A}.4$ and $\mathcal{A}.5$. Let Linear^- be the theory $\mathsf{Product}^-$ augmented with axiom schema $\mathcal{A}.6$. Let First^- be the theory Linear^- augmented with axiom schema $\mathcal{A}.7$.

Definition 1.4 (Axiom instance). An *instance of an axiom* $\mathcal{A}$ is the result of a consistent substitution of all the variables in $\mathcal{A}$ by type expressions of the general grammar.

For example, $\big(\mathsf{a} \rightarrow (\mathbf{T} \times \mathsf{b})\big) \times \mathsf{c} = \mathsf{c} \times \big(\mathsf{a} \rightarrow (\mathbf{T} \times \mathsf{b})\big)$ is an instance of the commutative axiom $\mathcal{A}.4$.

Definition 1.5 (Derivation sequence). Let Θ be a theory, e.g., $\Theta = \mathsf{Equality}$, or $\Theta = \mathsf{First}^-$. Then, the sequence $\tau_1 = \tau'_1, \ldots, \tau_m = \tau'_m$ is called a *derivation sequence* in Θ if for $i = 1, \ldots, m$, $\tau_i = \tau'_i$ is either an instance of an axiom in Θ or the result of applying one of the four inference rules on previous equalities. For types τ, τ' we write $\Theta \vdash \tau = \tau'$ when there exists a derivation sequence ending with the equality $\tau = \tau'$.

Let τ and τ' be two given types. We use the notation $\tau = \tau'$ as an abbreviation for $\mathsf{Equality} \vdash \tau = \tau'$.

Definition 1.6. *The first order isomorphism problem is to decide whether* $\mathsf{First} \vdash \tau = \tau'$.

The first order isomorphism problem has been known to be decidable for over a decade (Soloviev, 1983; Bruce et al., 1991). Previous to our work, the best known bound was $O(n^2 \log n)$ time using $O(n^2)$ space (Considine, 2000). *Our main result is in reducing the time to* $O(n \log^2 n)$ *time and the space to* $O(n)$.

One of the difficult issues in obtaining an efficient algorithm for the problem is dealing with the commutative and associative nature of product (axioms $\mathcal{A}.4$ and $\mathcal{A}.5$). Concentrating on this we define the product isomorphism problem.

Definition 1.7. *The product isomorphism problem is to decide whether* $\mathsf{Product} \vdash \tau = \tau'$.

We apply the standard abbreviation of using the $\prod$ symbol to denote (an associated to the left) product of several *terms*, i.e., for $k \geq 2$,

$$\prod_{i=1}^{k} \tau_i = \Big(\cdots \Big((\tau_1 \times \tau_2) \times \tau_3\Big) \cdots \times \tau_k\Big), \tag{1.2}$$

When the commutative and associative axioms apply, we shall write products without parenthesis. Consider, for example, the following product:

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

(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 the product isomorphism problem variant is to determine whether the above is isomorphic to

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

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

$$\mathsf{aaaaabbcdrr}. \tag{1.5}$$

In this paper we show that the product isomorphism problem is decidable in linear time.[‡] This result is based on the observation that it can be determined that (1.3) and (1.4) are isomorphic without using a super-linear sorting procedure, but rather by employing 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 (2000) gave an $O(n^2)$ time algorithm for a *recursive* product isomorphism problem, defined by the addition of a grammar rule $\tau ::= \mu\alpha.\tau$ where α is a type variable, and a folding/unfolding axiom

$$(\mathcal{A}.8) \qquad \mu\alpha.A = A[(\mu\alpha.A)/\alpha].$$

(As usual, the notation $A[B/\alpha]$ stands for a type expression A where each occurrence of α is replaced by B.) This result was later improved to $O(n \log n)$ time (Jha et al., 2002a) using a reduction to the problem of finding size-stable partitions of a directed graph.

We note that the recursive product isomorphism problem is not a simple a generalization of our product isomorphism problem. The reason is that isomorphism between recursive product-types should be defined in terms of their infinite unfoldings which are regular trees. To reason about these infinite structure, inductive variants of the *congruence of* $\times$ and *congruence of* $\rightarrow$ inference rules must be used. It was found (Palsberg, personal communication) that the combination of these variants with the folding/unfolding axiom and the unit axioms $\mathcal{A}.1$–$\mathcal{A}.3$. gives rise to an inconsistent system. These axioms were therefore omitted from the recursive product type systems. It remains a challenge to find a reformulation of the inference rules in Definition 1.1 which is consistent with all axioms $\mathcal{A}.1$–$\mathcal{A}.8$.

More difficult than the problem of product isomorphism is the problem variant defined by the Linear theory, which adds the currying axiom.

Definition 1.8. *The linear isomorphism problem is to decide whether* $\mathsf{Linear} \vdash \tau = \tau'$.

[‡] Jha (personal communication, September 2002) reports on independent discovery of an algorithm for this sub-problem, with similar complexity bounds, published in (Jha et al., 2002b).

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 (Andreev and Soloviev, 1997). Although not previously mentioned, both algorithms (Jha et al., 2002a; Considine, 2000) improve the running time to $O(n \log n)$. *We advance the state of the art by showing that linear isomorphism is also decidable in linear time.*

Linear isomorphism combined with the folding/unfolding axiom may generate products with an unbounded number of terms, which makes it difficult to apply the standard algorithms for recursive type isomorphisms. Consider, for example, the type

$$\mu\alpha.(\mathsf{a} \to \alpha). \tag{1.6}$$

The following equality is an instance of the folding/unfolding axiom

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

Repeated use of the folding/unfolding axiom proves that type (1.6) is isomorphic to

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

Finally, by using the currying axiom we can produce a product with any number of terms.

The final step toward solving the first order isomorphism problem is to deal with the distributive axiom $\mathcal{A}.7$. As we shall see, the difficulty in doing so is that a naive application of this axiom may lead to an exponential blowup of the input types.

1.3. *Intuition: 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. We assume that types use a standard expression-tree representation in memory, and that each *rule application* in the reduction system is implemented as a transformation of this data structure.

For example, the reduction system of Rittri (1990) has seven rules

$$\begin{array}{lrcl}
\mathcal{R}.1 & \mathbf{T} \times A & \Rightarrow & A \\
\mathcal{R}.2 & A \times \mathbf{T} & \Rightarrow & A \\
\mathcal{R}.3 & \mathbf{T} \to A & \Rightarrow & A \\
\mathcal{R}.4 & A \to \mathbf{T} & \Rightarrow & \mathbf{T} \\
\mathcal{R}.5 & A \times (B \times C) & \Rightarrow & (A \times B) \times C \\
\mathcal{R}.6 & A \to (B \to C) & \Rightarrow & (A \times B) \to C \\
\mathcal{R}.7 & A \to (B \times C) & \Rightarrow & (A \to B) \times (A \to C)
\end{array} \tag{1.7}$$

Rittri proved that the rules $\mathcal{R}.1$–$\mathcal{R}.7$ are confluent and terminating. Therefore, by repeated application of the rules the input types are reduced to a *normal form*.

In the degenerate case in which one or both of the inputs is reduced to $\mathbf{T}$, the input types are isomorphic if and only if they both reduce to $\mathbf{T}$. (This intuitive statement is given a formal proof in Section 2.) Otherwise, the normal forms do not contain the symbol $\mathbf{T}$. Furthermore, these rules can always simplify the structure of the right operand of $\to$, unless it is a primitive-type.

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 (1.7) (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

$$A \to (B \times C) \Rightarrow \begin{cases} (\alpha \to B) \times (\alpha \to C) \\ \alpha = A \end{cases}, \tag{1.8}$$

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

Rittri (1991) observed that using (1.8) ensures a polynomially sized representation of the normal form: Each application of transformation (1.8) 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):

$$\left(\mathsf{b}_1\left(\mathsf{b}_2\cdots\left(\mathsf{b}_{n-2}(\mathsf{b}_{n-1}\mathsf{b}_n^{\mathsf{a}_n})^{\mathsf{a}_{n-1}}\right)^{\mathsf{a}_{n-2}}\cdots\right)^{\mathsf{a}_2}\right)^{\mathsf{a}_1}, \tag{1.9}$$

whose normal form is

$$\mathsf{b}_1^{\mathsf{a}_1}\mathsf{b}_2^{\mathsf{a}_2\mathsf{a}_1}\cdots\mathsf{b}_{n-1}^{\mathsf{a}_{n-1}\cdots\mathsf{a}_1}\mathsf{b}_n^{\mathsf{a}_n\cdots\mathsf{a}_1}. \tag{1.10}$$

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

Remark 1.9. Deriving (1.9) starting at the outer-most parenthesis, yields the representation

$$\mathsf{b}_1^{\alpha_1}\cdots\mathsf{b}_n^{\alpha_n}, \tag{1.11}$$

where $\alpha_1 = \mathsf{a}_1$, and $\alpha_i = \mathsf{a}_i\alpha_{i-1}$ for $i = 2, \ldots, n$. Note that (1.11) requires only linear space whereas (1.10) is quadratic.

Having bounded 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 (2000) was able to reduce the runtime to polynomial. We should note that Considine's 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.

Considine's algorithm partitions all nodes in the directed acyclic graph (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 in this traversal was to determine whether product-nodes are isomorphic. Two key properties made Considine's $O(n^2 \log n)$ time and $O(n^2)$ 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 of no more than n terms.
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, thereby 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 of time to $O(n \log^2 n)$ are made possible by 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 1.9 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.

Road map Our algorithms employ four successive normal forms, all of which can be computed in linear time and space. Each normal form stands for a "simpler" isomorphic representation, obtained by exhaustively applying some of the rules (1.7).

The normal form $\mathrm{nf}_{\mathbf{T}}$, described in Section 2, is computed by applying rules $\mathcal{R}.1$–$\mathcal{R}.4$ to remove (essentially) all occurrences of $\mathbf{T}$. We further show in this section, that $\mathrm{nf}_{\mathbf{T}}$ makes it possible to completely ignore the unit axioms in the main algorithms.

The normal form nf_{c}, which takes care of the *currying* axiom, is the subject of Section 3, where we show how linear isomorphism can be reduced to product isomorphism.

To solve the product isomorphism problem, we need a procedure for comparing long products without sorting their terms. Section 4 develops this procedure as part of a general algorithm for multi-set partitioning. Section 5 then gives the concrete algorithm for the product isomorphism problem. In the algorithm the *associative* rule $\mathcal{R}.5$ is first applied to produce the normal form nf_{a}. The normalized types are then compared in a bottom-up traversal, while invoking the multi-set partitioning algorithm at each level.

Section 6 then shows how an exhaustive application of the *distributive* rule $\mathcal{R}.7$ produces the normal form nf_{d}. A linear space encoding for nf_{d}, called the $\mathbf{P}/\mathbf{F}$-graph, is also described in this section. Unexpanded products in the $\mathbf{P}/\mathbf{F}$-graph form a *tree structure*, such that each product inherits the terms of its parent. Section 7 employs multi-set partitioning in comparing unexpanded products in this tree structure. Section 8 fine-tunes 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 Section 9. Section 10 lists some open questions.

2. Eliminating Unit Types

This section describes a linear time and space algorithm for eliminating the unit axioms. Algorithm `EliminateUnits` receives as input two types: τ and τ', both conforming to the *general grammar*, describing arbitrary first order types.

General Grammar

$$\tau ::= \mathbf{T} \quad | \quad x \quad | \quad \tau \to \tau \quad | \quad \tau \times \tau.$$

The output comprises two types σ and σ', such that

$$\mathsf{First} \vdash \tau = \tau' \Leftrightarrow \mathsf{First}^{-} \vdash \sigma = \sigma'.$$

(At the end of this section we show that a similar claim can be made with regards to theories Linear and $\mathsf{Product}$.) The details are in Algorithm 1.

If either of τ or τ' is isomorphic to $\mathbf{T}$ then the algorithm returns a decision whether $\mathsf{First} \vdash \tau = \tau'$ (lines 4 and 6). Otherwise, i.e., when both τ and τ' are not isomorphic to $\mathbf{T}$, the algorithm returns two types σ and σ' such that $\mathsf{First} \vdash \tau = \tau' \Leftrightarrow \mathsf{First}^{-} \vdash \sigma = \sigma'$ (line 8). Both σ and σ' conform to the following *no-unit grammar*, in which the symbol $\mathbf{T}$ never occurs.

No-Unit Grammar

$$\tau ::= x \quad | \quad \tau \to \tau \quad | \quad \tau \times \tau$$

The crux of the algorithm is the transformation of the inputs into their normal form in lines 1 and 2. For a type τ, its normal form $\mathrm{nf}_{\mathbf{T}}(\tau)$ is a type isomorphic to τ, i.e., $\mathsf{First} \vdash \tau = \mathrm{nf}_{\mathbf{T}}(\tau)$, where $\mathrm{nf}_{\mathbf{T}}(\tau)$ is either the type $\mathbf{T}$ or it conforms to the no-unit grammar.

Algorithm 1 `EliminateUnits` (τ, τ')

Given two types τ and τ' conforming to the general grammar, return either (i) a decision whether First $\vdash \tau = \tau'$*, or (ii) a pair of two types σ, σ' conforming to the no-unit grammar such that* First $\vdash \tau = \tau' \Leftrightarrow$ First$^{-} \vdash \sigma = \sigma'$.

```
1: $\sigma \leftarrow \mathrm{nf}_{\mathbf{T}}(\tau)$
2: $\sigma' \leftarrow \mathrm{nf}_{\mathbf{T}}(\tau')$
3: If $\sigma = \mathbf{T}$ and $\sigma' = \mathbf{T}$ then
4:     return true // Types $\tau$ and $\tau'$ are isomorphic
5: else if $\sigma = \mathbf{T}$ or $\sigma' = \mathbf{T}$ then
6:     return false // Types $\tau$ and $\tau'$ are not isomorphic
7: else
8:     return $\langle \sigma, \sigma' \rangle$
9: fi
```

The following is an algorithmic definition of the normalizing function $\mathrm{nf}_{\mathbf{T}}$.[§] The function recursively traverses the tree representing the input type, while applying rules $\mathcal{R}.1$–$\mathcal{R}.4$ whenever possible.

$$\mathrm{nf}_{\mathbf{T}}(\tau) = \begin{cases} \mathbf{T} & \text{if } \tau = \mathbf{T} \\ x & \text{if } \tau = x \\ R_{1,2}(\mathrm{nf}_{\mathbf{T}}(\tau_a), \mathrm{nf}_{\mathbf{T}}(\tau_b)) & \text{if } \tau = \tau_a \times \tau_b \\ R_{3,4}(\mathrm{nf}_{\mathbf{T}}(\tau_a), \mathrm{nf}_{\mathbf{T}}(\tau_b)) & \text{if } \tau = \tau_a \to \tau_b \end{cases} \tag{2.1}$$

After the children of a node have been simplified by the recursive calls, function $\mathrm{nf}_{\mathbf{T}}$ may invoke, depending on the node type, one of two auxiliary functions to simplify the node itself. The first such function applies the product-unit rules ($\mathcal{R}.1$ and $\mathcal{R}.2$).

$$R_{1,2}(\sigma_a, \sigma_b) = \begin{cases} \sigma_b & \text{if } \sigma_a = \mathbf{T} \quad \text{// \textit{apply rule} } \mathcal{R}.1 \\ \sigma_a & \text{if } \sigma_b = \mathbf{T} \quad \text{// \textit{apply rule} } \mathcal{R}.2 \\ \sigma_a \times \sigma_b & \text{otherwise} \end{cases} \tag{2.2}$$

The other auxiliary function applies the function-unit rules ($\mathcal{R}.3$ and $\mathcal{R}.4$).

$$R_{3,4}(\sigma_a, \sigma_b) = \begin{cases} \sigma_b & \text{if } \sigma_a = \mathbf{T} \quad \text{// \textit{apply rule} } \mathcal{R}.3 \\ \mathbf{T} & \text{if } \sigma_b = \mathbf{T} \quad \text{// \textit{apply rule} } \mathcal{R}.4 \\ \sigma_a \to \sigma_b & \text{otherwise} \end{cases} \tag{2.3}$$

Let $|\tau|$ denote the *size* of a type τ, defined as the number of nodes in the standard abstract syntax tree representation of τ. Many of our proofs employ *structural induction* which is essentially induction on the input size. In the inductive step, we shall rely on the *type decomposability property*: if $|\tau| > 1$ (i.e., $\tau \neq x$ and $\tau \neq \mathbf{T}$) then τ is represented as a type-operator node with two children representing types τ_a and τ_b, such that $|\tau| = |\tau_a| + |\tau_b| + 1$.

[§] Here and henceforth, we use the same notation for the *normal form*, and for the (algorithmic) function which given a type, generates and returns its normal form. No confusion should arise as a result of this overloading.

Lemma 2.1. Let τ be a type which conforms to the general grammar, and let $\sigma = \mathrm{nf}_{\mathbf{T}}(\tau)$. Then, **(i)** the invocation $\mathrm{nf}_{\mathbf{T}}(\tau)$ requires $O(|\tau|)$ time, **(ii)** $|\sigma| \leq |\tau|$, **(iii)** $\sigma = \mathbf{T}$ or σ conforms to the no-unit grammar, and **(iv)** $\mathsf{Product} \vdash \tau = \sigma$.

PROOF. All parts are proved by structural induction. The inductive base, $|\tau| = 1$, is covered by the first two cases ($\tau = x$ and $\tau = \mathbf{T}$) in (2.1). Both these cases execute in constant time, and their output is identical to their input. Moreover, this output either conforms to the no-unit grammar or is $\mathbf{T}$.

In proving the inductive step we use the inductive hypothesis and the decomposability property. For **(i)** we note that only a constant amount of work is carried out prior to and after the recursive calls (i.e., in $R_{1,2}$ and $R_{3,4}$). Noting that $R_{1,2}$ and $R_{3,4}$ do not create new nodes proves the inductive step of **(ii)**. The inductive step of **(iii)** is carried out by checking that the output of $R_{1,2}$ and $R_{3,4}$ satisfies **(iii)** whenever their input does. Part **(iv)** is proved by noting that functions $R_{1,2}$ and $R_{3,4}$ only apply rules conforming to the axioms $\mathcal{A}.1$–$\mathcal{A}.4$. $\square$

Lemma 2.1 proves the correctness of Algorithm 1 in the cases it terminates in line 4. Next we would like to prove that when the algorithm terminates in line 6 then τ and τ' are indeed not isomorphic. Note that the algorithm terminates in line 6 if and only if either $\sigma = \mathbf{T}$ and $\sigma' \neq \mathbf{T}$ or the reverse. Therefore we must prove that $\mathbf{T}$ cannot be isomorphic to any type σ which conforms to the no-unit grammar. We will use the technique of abstract interpretation (Cousot and Cousot, 1992) for doing so.

For a type τ define the abstract interpretation function $\mathrm{is}_{\mathbf{T}}(\tau)$ as follows

$$\mathrm{is}_{\mathbf{T}}(\tau) = \begin{cases} 1 & \text{if } \tau = \mathbf{T} \\ 0 & \text{if } \tau = x \\ \mathrm{is}_{\mathbf{T}}(\tau_a) \cdot \mathrm{is}_{\mathbf{T}}(\tau_b) & \text{if } \tau = \tau_a \times \tau_b \\ \mathrm{is}_{\mathbf{T}}(\tau_b) & \text{if } \tau = \tau_a \rightarrow \tau_b \end{cases} \tag{2.4}$$

Note that $\mathrm{is}_{\mathbf{T}}(\tau)$ returns either 0 or 1. We next prove that $\mathrm{is}_{\mathbf{T}}(\tau)$ is 1 precisely when $\mathrm{nf}_{\mathbf{T}}(\tau) = \mathbf{T}$ (hence the name $\mathrm{is}_{\mathbf{T}}$).

Lemma 2.2. $\mathrm{nf}_{\mathbf{T}}(\tau) = \mathbf{T} \Leftrightarrow \mathrm{is}_{\mathbf{T}}(\tau) = 1$.

PROOF. By examining the definitions of $\mathrm{nf}_{\mathbf{T}}$, $R_{1,2}$ and $R_{3,4}$ we see that $\mathrm{nf}_{\mathbf{T}}(\tau) = \mathbf{T}$ if and only if one of the following holds

1. $\tau = \mathbf{T}$.
2. $\tau = \tau_a \times \tau_b$, where $\mathrm{nf}_{\mathbf{T}}(\tau_a) = \mathbf{T}$ and $\mathrm{nf}_{\mathbf{T}}(\tau_b) = \mathbf{T}$.
3. $\tau = \tau_a \rightarrow \tau_b$, where $\mathrm{nf}_{\mathbf{T}}(\tau_b) = \mathbf{T}$.

Therefore $\mathrm{nf}_{\mathbf{T}}(\tau) = \mathbf{T}$ if and only if $\mathrm{is}_{\mathbf{T}}(\tau) = 1$. $\square$

Lemma 2.3. $\mathsf{First} \vdash \tau = \tau' \Rightarrow \mathrm{is}_{\mathbf{T}}(\tau) = \mathrm{is}_{\mathbf{T}}(\tau')$

PROOF. By induction on the length of the derivation sequence of $\mathsf{First} \vdash \tau = \tau'$. Recall that each equality in the derivation sequence is either an instance of an axiom or an application of one of the inference rules on previous equalities.

The induction base is that there is precisely one such equality $\tau = \tau'$, which must be an instance of an axiom $\mathcal{A}.0, \ldots, \mathcal{A}.7$. We can easily check in each of the axioms that $\mathrm{is}_{\mathbf{T}}(\tau) =$

$\mathrm{is}_{\mathbf{T}}(\tau')$. For example, if $\tau = \tau'$ is an instance of $\mathcal{A}.7$. then $\tau = \tau_a \to (\tau_b \times \tau_c)$ and $\tau' = (\tau_a \to \tau_b) \times (\tau_a \to \tau_c)$. We have

$$\mathrm{is}_{\mathbf{T}}(\tau) = \mathrm{is}_{\mathbf{T}}\big(\tau_a \to (\tau_b \times \tau_c)\big) = \mathrm{is}_{\mathbf{T}}(\tau_b \times \tau_c) = \mathrm{is}_{\mathbf{T}}(\tau_b) \cdot \mathrm{is}_{\mathbf{T}}(\tau_c),$$

and

$$\mathrm{is}_{\mathbf{T}}(\tau') = \mathrm{is}_{\mathbf{T}}\big((\tau_a \to \tau_b) \times (\tau_a \to \tau_c)\big) = \mathrm{is}_{\mathbf{T}}(\tau_a \to \tau_b) \cdot \mathrm{is}_{\mathbf{T}}(\tau_a \to \tau_c) = \mathrm{is}_{\mathbf{T}}(\tau_b) \cdot \mathrm{is}_{\mathbf{T}}(\tau_c).$$

To prove the induction step we examine the last step of the derivation sequence. If this step is an axiom instance, then the same considerations as in the induction base apply. Otherwise one of the following inference rules was applied: symmetry, transitivity, congruence of $\times$, or congruence of $\to$. We can easily check each of inference rules by using the inductive hypothesis. For instance, suppose that the congruence rule of $\times$ was applied:

$$\frac{\tau_a = \tau_b, \tau_c = \tau_d}{\tau_a \times \tau_c = \tau_b \times \tau_d}.$$

By the inductive hypothesis, we have that $\mathrm{is}_{\mathbf{T}}(\tau_a) = \mathrm{is}_{\mathbf{T}}(\tau_b)$ and $\mathrm{is}_{\mathbf{T}}(\tau_c) = \mathrm{is}_{\mathbf{T}}(\tau_d)$. Therefore, we can deduce that

$$\mathrm{is}_{\mathbf{T}}(\tau_a \times \tau_c) = \mathrm{is}_{\mathbf{T}}(\tau_a) \cdot \mathrm{is}_{\mathbf{T}}(\tau_c) = \mathrm{is}_{\mathbf{T}}(\tau_b) \cdot \mathrm{is}_{\mathbf{T}}(\tau_d) = \mathrm{is}_{\mathbf{T}}(\tau_b \times \tau_d).$$

□

Corollary 2.4. Let σ be a type conforming to the no-unit grammar. Then σ is not isomorphic to $\mathbf{T}$, i.e., $\mathsf{First} \nvdash \sigma = \mathbf{T}$.

PROOF. Assume by contradiction that $\mathsf{First} \vdash \sigma = \mathbf{T}$. Then, by Lemma 2.3, $\mathrm{is}_{\mathbf{T}}(\sigma) = \mathrm{is}_{\mathbf{T}}(\mathbf{T})$. Since σ conforms to the no-unit grammar, we have that $\mathrm{is}_{\mathbf{T}}(\sigma) = 0$, which contradicts the fact that $\mathrm{is}_{\mathbf{T}}(\mathbf{T}) = 1$. □

Finally, we will prove the correctness of Algorithm 1 in the cases it terminates in line 8, i.e., we need to show that

$$\mathsf{First} \vdash \tau = \tau' \Leftrightarrow \mathsf{First}^- \vdash \mathrm{nf}_{\mathbf{T}}(\tau) = \mathrm{nf}_{\mathbf{T}}(\tau').$$

The $\Leftarrow$ direction follows directly from Lemma 2.1**(iv)** combined with the facts that $\mathsf{First}^- \subseteq \mathsf{First}$ and $\mathsf{Product} \subseteq \mathsf{First}$.

Lemma 2.5. Let τ and τ' be two types conforming to the general grammar. Then,

$$\mathsf{First} \vdash \tau = \tau' \Rightarrow \mathsf{First}^- \vdash \mathrm{nf}_{\mathbf{T}}(\tau) = \mathrm{nf}_{\mathbf{T}}(\tau').$$

PROOF. By induction on the length of the derivation sequence of $\mathsf{First} \vdash \tau = \tau'$, whose final step must be the equality $\tau = \tau'$. In the induction base, this equality must be instance of one of the axioms $\mathcal{A}.0, \ldots, \mathcal{A}.7$. If $\tau = \tau'$ is an instance of $\mathcal{A}.3$, then $\tau = \mathbf{T} \to \tau_a$ and $\tau' = \tau_a$. We see that $\mathrm{nf}_{\mathbf{T}}(\tau) = \mathrm{nf}_{\mathbf{T}}(\tau')$, and hence $\mathsf{First}^- \vdash \mathrm{nf}_{\mathbf{T}}(\tau) = \mathrm{nf}_{\mathbf{T}}(\tau')$. A similar consideration and conclusion applies if $\tau = \tau'$ is an instance of axioms $\mathcal{A}.0$–$\mathcal{A}.2$

Suppose that $\tau = \tau'$ is an instance of the commutative axiom $\mathcal{A}.4$, i.e., $\tau = \tau_a \times \tau_b$ and $\tau' = \tau_b \times \tau_a$. We have

$$\mathrm{nf}_{\mathbf{T}}(\tau) = R_{1,2}(\mathrm{nf}_{\mathbf{T}}(\tau_a), \mathrm{nf}_{\mathbf{T}}(\tau_b)),$$
$$\mathrm{nf}_{\mathbf{T}}(\tau') = R_{1,2}(\mathrm{nf}_{\mathbf{T}}(\tau_b), \mathrm{nf}_{\mathbf{T}}(\tau_a)).$$

If either $\mathrm{nf}_{\mathbf{T}}(\tau_a) = \mathbf{T}$ or $\mathrm{nf}_{\mathbf{T}}(\tau_b) = \mathbf{T}$ then $\mathrm{nf}_{\mathbf{T}}(\tau) = \mathrm{nf}_{\mathbf{T}}(\tau')$, therefore $\mathsf{First}^- \vdash \mathrm{nf}_{\mathbf{T}}(\tau) = \mathrm{nf}_{\mathbf{T}}(\tau')$. Otherwise

$$\begin{aligned} \mathrm{nf}_{\mathbf{T}}(\tau) &= \mathrm{nf}_{\mathbf{T}}(\tau_a) \times \mathrm{nf}_{\mathbf{T}}(\tau_b), \\ \mathrm{nf}_{\mathbf{T}}(\tau') &= \mathrm{nf}_{\mathbf{T}}(\tau_b) \times \mathrm{nf}_{\mathbf{T}}(\tau_a), \end{aligned}$$

and the commutative axiom $\mathcal{A}.4$ proves that $\mathsf{First}^- \vdash \mathrm{nf}_{\mathbf{T}}(\tau) = \mathrm{nf}_{\mathbf{T}}(\tau')$. A similar, though more laborious, consideration proves the same induction base in the case that $\tau = \tau'$ is an instance of $\mathcal{A}.5$–$\mathcal{A}.7$.

In the induction step, we focus on the case that the final equality was obtained by one of the inference rules: symmetry, transitivity, congruence of $\times$, or congruence of $\rightarrow$. (The case that this equality is an axiom instance is identical to the induction base.)

Consider, for instance, the inference rule for congruence of $\times$. Then $\tau = \tau_a \times \tau_b$ and $\tau' = \tau_c \times \tau_d$. The inductive hypothesis is that $\mathsf{First}^- \vdash \mathrm{nf}_{\mathbf{T}}(\tau_a) = \mathrm{nf}_{\mathbf{T}}(\tau_c)$ and $\mathsf{First}^- \vdash \mathrm{nf}_{\mathbf{T}}(\tau_b) = \mathrm{nf}_{\mathbf{T}}(\tau_d)$. We need to show that $\mathsf{First}^- \vdash \mathrm{nf}_{\mathbf{T}}(\tau_a \times \tau_b) = \mathrm{nf}_{\mathbf{T}}(\tau_c \times \tau_d)$, or in other words, that

$$\mathsf{First}^- \vdash R_{1,2}\big(\mathrm{nf}_{\mathbf{T}}(\tau_a), \mathrm{nf}_{\mathbf{T}}(\tau_b)\big) = R_{1,2}\big(\mathrm{nf}_{\mathbf{T}}(\tau_c), \mathrm{nf}_{\mathbf{T}}(\tau_d)\big). \tag{2.5}$$

Examining definition (2.2) of $R_{1,2}$ we see that the proof must distinguish between several cases, depending on whether the arguments to this function are $\mathbf{T}$.

To make this distinction, we apply Lemma 2.3, obtaining that $\mathrm{nf}_{\mathbf{T}}(\tau_a) = \mathbf{T}$ if and only if $\mathrm{nf}_{\mathbf{T}}(\tau_c) = \mathbf{T}$, and $\mathrm{nf}_{\mathbf{T}}(\tau_b) = \mathbf{T}$ if and only if $\mathrm{nf}_{\mathbf{T}}(\tau_d) = \mathbf{T}$. (The lemma condition is met by the inductive hypothesis and the fact that $\mathsf{First}^- \subseteq \mathsf{First}$.)

Consider the case that $\mathrm{nf}_{\mathbf{T}}(\tau_a) \neq \mathbf{T}$ and $\mathrm{nf}_{\mathbf{T}}(\tau_b) \neq \mathbf{T}$. Then, (2.5) takes the form

$$\mathsf{First}^- \vdash \mathrm{nf}_{\mathbf{T}}(\tau_a) \times \mathrm{nf}_{\mathbf{T}}(\tau_b) = \mathrm{nf}_{\mathbf{T}}(\tau_c) \times \mathrm{nf}_{\mathbf{T}}(\tau_d).$$

The derivation sequence for this can be obtained by concatenating the derivation sequences of the inductive hypothesis and a single application of the congruence of $\times$ inference rule. The other cases of (2.5) are simpler, since the desired derivation sequence is one of those of the inductive hypothesis.

The induction step in the case the final equation is an instance of any of the other inference rules is carried out similarly. $\square$

It is straightforward to check that if σ conforms to the no-unit grammar, then $\mathrm{nf}_{\mathbf{T}}(\sigma) = \sigma$. We therefore have:

Corollary 2.6. Suppose that both τ and τ' conform to the no-unit grammar. Then,

$$\mathsf{First} \vdash \tau = \tau' \Leftrightarrow \mathsf{First}^- \vdash \tau = \tau'.$$

Much in the same fashion we can show

Corollary 2.7. Suppose that both τ and τ' conform to the no-unit grammar. Then,

$$\begin{aligned} \mathsf{Linear} \vdash \tau = \tau' &\Leftrightarrow \mathsf{Linear}^- \vdash \tau = \tau', \\ \mathsf{Product} \vdash \tau = \tau' &\Leftrightarrow \mathsf{Product}^- \vdash \tau = \tau'. \end{aligned}$$

3. Reduction of Linear Isomorphism to Product Isomorphism

In this section we show a linear time and space reduction of linear isomorphism to product isomorphism. The inputs are two types τ and τ' conforming to the no-unit grammar. The algorithm outputs are two types σ, σ' such that

$$\mathsf{Linear}^- \vdash \tau = \tau' \Leftrightarrow \mathsf{Product}^- \vdash \sigma = \sigma'.$$

Noting that Linear^- adds to $\mathsf{Product}^-$ the currying axiom ($\mathcal{A}.6$), the algorithm converts the inputs τ and τ' into a normal form in which all curried functions are brought into an equivalent un-curried representation. This is achieved by recursively applying the anti-currying rule $\mathcal{R}.6$ to τ and τ'. The result then conforms to the un-curried grammar, in which the pattern $A \to (B \to C)$ is not allowed.

Un-curried Grammar

$$\tau ::= x \quad | \quad \tau \to x \quad | \quad \tau \to (\tau \times \tau) \quad | \quad \tau \times \tau$$

Algorithmically, the normal form is computed using function $\mathrm{nf_c}$.

$$\mathrm{nf_c}(\tau) = \begin{cases} x & \text{if } \tau = x \\ \mathrm{nf_c}(\tau_a) \times \mathrm{nf_c}(\tau_b) & \text{if } \tau = \tau_a \times \tau_b \\ R_6(\mathrm{nf_c}(\tau_a), \mathrm{nf_c}(\tau_b)) & \text{if } \tau = \tau_a \to \tau_b \end{cases} \tag{3.1}$$

If a node represents a function-type, then function R_6 checks whether the return type of this function is another function type, and if so, applies the anti-currying rule.

$$R_6(\sigma_a, \sigma_b) = \begin{cases} (\sigma_a \times \sigma_1) \to \sigma_2 & \text{if } \sigma_b = \sigma_1 \to \sigma_2 \qquad \textit{// apply rule } \mathcal{R}.6 \\ \sigma_a \to \sigma_b & \text{otherwise} \end{cases} \tag{3.2}$$

Lemma 3.1. Let τ be a type conforming to the no-unit grammar, and let $\sigma = \mathrm{nf_c}(\tau)$. Then, **(i)** the call $\mathrm{nf_c}(\tau)$ executes in $O(|\tau|)$ time; **(ii)** $\mathsf{Linear}^- \vdash \tau = \sigma$; **(iii)** $|\sigma| = |\tau|$; and **(iv)** type σ conforms to the un-curried grammar.

PROOF. Parts **(i)**, **(ii)**, and **(iii)** are proved by structural induction, following the outline of the proof of Lemma 2.1.

In proving **(iv)** we note that there are two restrictions in the un-curried grammar. The first is that there are no occurrences of $\mathbf{T}$. This follows from the assumption that τ conforms to the no-unit grammar.

The second restriction is that the return type of all function-types is not a function-type. We show that $\mathrm{nf_c}(\tau)$ conforms to this restriction by induction on the depth of the recursive calls of $\mathrm{nf_c}$. The inductive base is the first case of (3.1) and is trivial. In the inductive step we must show that the return type of a function cannot be a function itself. A node corresponding to a function-type can be generated by $\mathrm{nf_c}$ only in the third case of (3.1). This node itself is generated by the invocation $R_6(\sigma_a, \sigma_b)$. Examining (3.2) we see that the return type of this node is σ_b precisely when σ_b is not a function-type. If however σ_b is a function-type, i.e., $\sigma_b = \sigma_1 \to \sigma_2$, then recall that σ_b was computed by a recursive application of $\mathrm{nf_c}$. Therefore, by the inductive hypothesis, σ_2, the return type of the current node is not a function-type. $\square$

It follows from Lemma 3.1**(ii)** that if the normal forms $\mathrm{nf_c}(\tau)$ and $\mathrm{nf_c}(\tau')$ are isomorphic by applications of the commutative and associative axioms, then τ and τ' are also isomorphic by application of the commutative, associative and currying axioms, i.e.,

$$\textsf{Product}^- \vdash \mathrm{nf_c}(\tau) = \mathrm{nf_c}(\tau') \Rightarrow \textsf{Linear}^- \vdash \tau = \tau'. \tag{3.3}$$

The remainder of this section is dedicated to proving the converse, i.e., that after the types where brought to their un-curried normal form, all that is required in deciding isomorphism is to apply the commutative and associative axioms. The proof is similar in spirit to that of Andreev and Soloviev (1997).

Lemma 3.2. $\textsf{Linear}^- \vdash \tau = \tau' \Rightarrow \textsf{Product}^- \vdash \mathrm{nf_c}(\tau) = \mathrm{nf_c}(\tau')$.

PROOF. The proof is by induction on the length of the derivation sequence of $\textsf{Linear}^- \vdash \tau = \tau'$, and follows the same outline as the proof of Lemma 2.3.

The induction base is that $\tau = \tau'$ is an instance of an axiom $\mathcal{A}.0, \ldots, \mathcal{A}.6$. This cannot be one of the unit axioms $\mathcal{A}.1, \ldots, \mathcal{A}.3$ since by assumption $\mathbf{T}$ does not occur in the input. In the case that the reflexive axiom ($\mathcal{A}.0$) was applied, it is trivial to see that $\mathrm{nf_c}(\tau) = \mathrm{nf_c}(\tau')$.

In the case that this axiom was the commutative axiom ($\mathcal{A}.4$), then $\tau = \tau_a \times \tau_b$ and $\tau' = \tau_b \times \tau_a$. It is easy to see that $\mathrm{nf_c}(\tau) = \mathrm{nf_c}(\tau_a) \times \mathrm{nf_c}(\tau_b)$ and $\mathrm{nf_c}(\tau') = \mathrm{nf_c}(\tau_b) \times \mathrm{nf_c}(\tau_a)$. Therefore, $\textsf{Product}^- \vdash \mathrm{nf_c}(\tau) = \mathrm{nf_c}(\tau')$. Similar consideration apply when this axiom was the associative axiom ($\mathcal{A}.5$).

The last axiom to consider is the currying axiom $\mathcal{A}.6$. In this case $\tau = (\tau_a \times \tau_b) \rightarrow \rho$ and $\tau' = \tau_a \rightarrow (\tau_b \rightarrow \rho)$. There are two cases to consider:

1. *Type ρ is not a function-type.* Examining the definitions (3.1) and (3.2), we find that
$$\mathrm{nf_c}(\tau) = \mathrm{nf_c}(\tau') = \Big[\mathrm{nf_c}(\tau_a) \times \mathrm{nf_c}(\tau_b)\Big] \rightarrow \mathrm{nf_c}(\rho).$$
2. *Type ρ is a function-type.* In this case we find the maximal k such that ρ can be written as
$$\rho = \rho_1 \rightarrow \big(\rho_2 \rightarrow \cdots (\rho_{k-1} \rightarrow \rho_k) \cdots\big).$$
Note that, by definition, ρ_k is not a function-type. Let
$$\varrho = \mathrm{nf_c}(\rho_1) \times \Big(\mathrm{nf_c}(\rho_2) \times \cdots \times \big(\mathrm{nf_c}(\rho_{k-2}) \times \mathrm{nf_c}(\rho_{k-1})\big) \cdots\Big).$$
It is then easy to check that
$$\begin{aligned}\mathrm{nf_c}(\tau) &= \Big[\big(\mathrm{nf_c}(\tau_a) \times \mathrm{nf_c}(\tau_b)\big) \times \varrho\Big] \rightarrow \mathrm{nf_c}(\rho_k),\\ \mathrm{nf_c}(\tau') &= \Big[\mathrm{nf_c}(\tau_a) \times \big(\mathrm{nf_c}(\tau_b) \times \varrho\big)\Big] \rightarrow \mathrm{nf_c}(\rho_k).\end{aligned}$$

In both cases we have that $\textsf{Product}^- \vdash \mathrm{nf_c}(\tau) = \mathrm{nf_c}(\tau')$.

To prove the induction step we examine the last step of the derivation sequence. If this step is an axiom instance, then the same considerations as in the induction base apply. Otherwise one of the following inference rules was applied: symmetry, transitivity, congruence of $\times$, or congruence of $\rightarrow$. The only difficulty is with the congruence rule of $\rightarrow$. Consider an instance of this inference rule:

$$\frac{\tau_a = \tau_b, \tau_c = \tau_d}{\tau_a \rightarrow \tau_c = \tau_b \rightarrow \tau_d}.$$

By the inductive hypothesis, we have that $\mathsf{Product}^- \vdash \mathrm{nf_c}(\tau_a) = \mathrm{nf_c}(\tau_b)$ and $\mathsf{Product}^- \vdash \mathrm{nf_c}(\tau_c) = \mathrm{nf_c}(\tau_d)$. We would like to prove that $\mathsf{Product}^- \vdash \mathrm{nf_c}(\tau_a \to \tau_c) = \mathrm{nf_c}(\tau_b \to \tau_d)$.

Note that since $\mathsf{Product}^- \vdash \mathrm{nf_c}(\tau_c) = \mathrm{nf_c}(\tau_d)$, their root nodes have the same type, i.e., both $\mathrm{nf_c}(\tau_c)$ and $\mathrm{nf_c}(\tau_d)$ are product-types, function-types, or primitive-types. There are two cases to consider:

1. *Types* $\mathrm{nf_c}(\tau_c)$ *and* $\mathrm{nf_c}(\tau_d)$ *are both not function-types.* We find that
$$\mathrm{nf_c}(\tau_a \to \tau_c) = \mathrm{nf_c}(\tau_a) \to \mathrm{nf_c}(\tau_c),$$
$$\mathrm{nf_c}(\tau_b \to \tau_d) = \mathrm{nf_c}(\tau_b) \to \mathrm{nf_c}(\tau_d).$$
2. *Types* $\mathrm{nf_c}(\tau_c)$ *and* $\mathrm{nf_c}(\tau_d)$ *are both function-types.* Let $\mathrm{nf_c}(\tau_c) = \varrho \to \rho$ and $\mathrm{nf_c}(\tau_d) = \varrho' \to \rho'$. Since $\mathsf{Product}^- \vdash \mathrm{nf_c}(\tau_c) = \mathrm{nf_c}(\tau_d)$ we have that $\mathsf{Product}^- \vdash \varrho = \varrho'$ and $\mathsf{Product}^- \vdash \rho = \rho'$. It is then easy to check that
$$\mathrm{nf_c}(\tau_a \to \tau_c) = \left[\mathrm{nf_c}(\tau_a) \times \varrho\right] \to \mathrm{nf_c}(\rho),$$
$$\mathrm{nf_c}(\tau_b \to \tau_d) = \left[\mathrm{nf_c}(\tau_b) \times \varrho'\right] \to \mathrm{nf_c}(\rho').$$

In both cases we have that $\mathsf{Product}^- \vdash \mathrm{nf_c}(\tau_a \to \tau_c) = \mathrm{nf_c}(\tau_b \to \tau_d)$. $\square$

4. 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. Cai and Paige (1995) review other linear-time algorithms for partitioning multi-sets.

Definition 4.1 (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 $a_{i_1} = \cdots = a_{i_m}$. The partitioning into equivalence classes is thus represented as a list of lists of indices.

Lemma 4.2. 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. $\square$

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

Definition 4.3 (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 4.4 (Renaming). *Let $\mathcal{U}$ be an arbitrary domain and let $\Gamma \subseteq \mathcal{U}$, $|\Gamma| = n$. Then a partial function $\Omega : \mathcal{U} \mapsto [1, n]$ is a* renaming *of Γ if Ω is defined on Γ and for any $a, b \in \Gamma$,*

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

Algorithm 2 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 2 `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:  $\Omega \leftarrow$ new int[U] // An uninitialized array of size U
2:  $\mho \leftarrow$ new int[n] // The inverse mapping of $\Omega$
3:  $\ell \leftarrow 0$ // $\ell$ is the current number of distinct values in the input
4:  For $i = 1, \ldots, n$ do // Compute $\Omega[a_i]$
5:     $t \leftarrow \Omega[a_i]$ // $t$ may be arbitrary if the value of $a_i$ is new
6:     If $1 \leq t \leq \ell$ andalso $\mho[t] = a_i$ then
7:        next $i$ // No new mapping since $a_i = a_j$ for some $j < i$
8:     else // Create a new mapping entry
9:        $\ell \leftarrow \ell + 1$ // A new distinct input value
10:       $\Omega[a_i] \leftarrow \ell$ // Store the mapping entry
11:       $\mho[\ell] \leftarrow a_i$ // Record the inverse pointer
12:    fi
13: od
```

Renaming makes it possible to generalize Lemma 4.2.

Lemma 4.5. Broad integer partitioning can be solved in $O(n)$ time and $O(U + n)$ space.

PROOF. After applying Algorithm 2, 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 4.6 (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 4.7. 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. $\square$

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 4.7 can be generalized further.

Lemma 4.8 (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 4.7, 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. $\square$

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 4.9 (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 4.10 (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 4.11. 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, Algorithm 2 is invoked to rename all integers in the input to fit the range $[1, n]$. The next step is to sort the multi-sets. However, if each of these is sorted independently the running time would not be linear. Instead, we concatenate the sets together, prefixing each input integer with the identifier of its multi-set. All the multi-sets can then be sorted by a single application of a radix sort.

We stress that we sort the *renamed* integers, not the initial multi-sets. This process is known as *weak sort* (Paige, 1994). Weak sort 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 4.8 in each partition. □

5. An Algorithm for the Product Isomorphism Problem

After units are eliminated, product isomorphism theory has only the commutative and associative axioms. These axioms allow products to be reordered until the two types match. Thus product isomorphism is in essence a series of multi-set partitioning problems. In this section we use the algorithms described in the previous section for these problems in developing an $O(n)$ time and space algorithm for product isomorphism. This algorithm receives two types, τ and τ', conforming to the no-unit grammar, and determines whether $\mathsf{Product}^{-} \vdash \tau = \tau'$.

The algorithm begins by *flattening all products* in the input, so that it conforms to the following product grammar.

Product Grammar

$$\rho ::= \prod_{i=1}^{k} \sigma \qquad\qquad (k \geq 1)$$

$$\sigma ::= x \quad | \quad \rho \to \rho$$

Note that we have extended the $\prod$ convention (1.2) to include products with a single term. Thus, in this grammar

$$\prod(x) = x. \tag{5.1}$$

Recall that by assumption the input cannot be isomorphic to $\mathbf{T}$, hence the start symbol ρ denotes products of at least one term. Each of these terms is either a primitive-type or a function-type.

Consider, for example, the following type, which will serve as a running example,

$$\big((\mathsf{a} \times \mathsf{b}) \to \mathsf{c}\big) \to \Big(\big(\mathsf{d} \times (\mathsf{e} \times \mathsf{f})\big) \times \big(\mathsf{g} \to (\mathsf{h} \times \mathsf{i})\big)\Big). \tag{5.2}$$

Figure 5.1 shows the expression tree of this type before and after flattening.

Algorithmically, the flattening process is carried out by computing the normal form defined by the function nf_{a}. This function receives a type τ conforming to the no-unit grammar, and exhaustively applies the associative rule $\mathcal{R}.5$. The output is a type conforming to the product

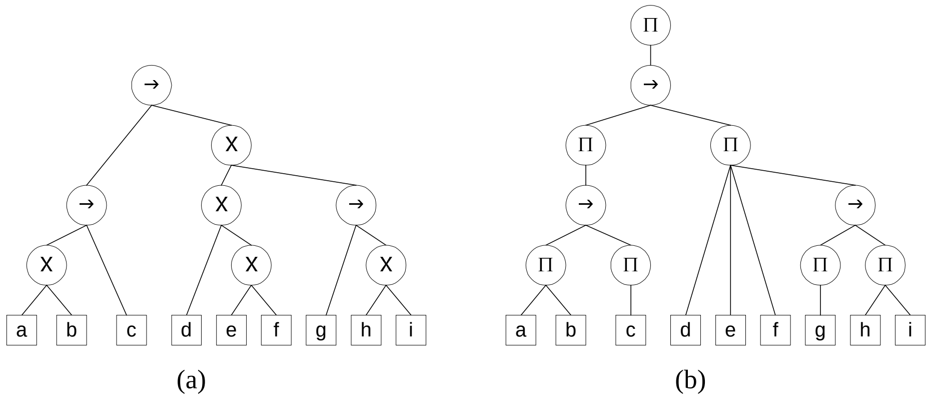


Fig. 5.1. An abstract syntax tree of type (5.2) before (a) and after (b) flattening

grammar.

$$\mathrm{nf_a}(\tau) = \begin{cases} \prod(x) & \text{if } \tau = x \\ \prod(\mathrm{nf_a}(\tau_a) \rightarrow \mathrm{nf_a}(\tau_b)) & \text{if } \tau = \tau_a \rightarrow \tau_b \\ \mathrm{nf_a}(\tau_a) \bowtie \mathrm{nf_a}(\tau_b) & \text{if } \tau = \tau_a \times \tau_b \qquad \textit{// apply rule } \mathcal{R}.5 \end{cases} \tag{5.3}$$

The operation $\bowtie$ denotes the concatenation of the terms of two products, i.e.,

$$\prod_{i=1}^{k'} \tau_i \bowtie \prod_{i=k'+1}^{k} \tau_i = \prod_{i=1}^{k} \tau_i .$$

Lemma 5.1. Let τ be a type conforming to the no-unit grammar, and let $\sigma = \mathrm{nf_a}(\tau)$. Then, **(i)** the call $\mathrm{nf_a}(\tau)$ executes in $O(|\tau|)$ time; **(ii)** $|\sigma| \leq 2|\tau|$; **(iii)** type σ conforms to the products grammar; and **(iv)** $\mathsf{Product}^- \vdash \tau = \sigma$

PROOF. Trivial by structural induction. Part **(iv)** is proved by interpreting $\prod$ nodes with conventions (1.2) and (5.1) and noting that only the associative rule $\mathcal{R}.5$ was applied in the definition of $\mathrm{nf_a}(\tau)$. $\square$

The flattened representation makes it easier to decide product isomorphism. The following lemma shows how this decision might be carried out.

Lemma 5.2. Let τ and τ' be two types conforming to the product grammar. Then, $\mathsf{Product}^- \vdash \tau = \tau'$ if and only if one of the following three statements holds:

1. Types τ and τ' are equal to the same primitive-type x.
2. Types τ and τ' are function-types, i.e., $\tau = \rho_1 \rightarrow \rho_2$ and $\tau' = \rho'_1 \rightarrow \rho'_2$, and $\mathsf{Product}^- \vdash \rho_1 = \rho'_1$ and $\mathsf{Product}^- \vdash \rho_2 = \rho'_2$.
3. Types τ and τ' are product-types with the same number of terms, i.e., $\tau = \prod_{i=1}^{k} \sigma_i$ and $\tau' = \prod_{i=1}^{k} \sigma'_i$, and there exists a bijection $\pi : [1,k] \mapsto [1,k]$, such that $\mathsf{Product}^- \vdash \sigma_i = \sigma'_{\pi(i)}$ for all i, $1 \leq i \leq k$.

PROOF. Direction $\Leftarrow$ is trivial. Direction $\Rightarrow$ is done by induction on the length of the derivation sequence of $\mathsf{Product}^- \vdash \tau = \tau'$. $\square$

The product grammar 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 product (function) types are always represented by nodes of odd (even) height.

Definition 5.3 (Height). *Let τ be a type conforming to the product grammar. Then, the height of a type, denoted $h(\tau)$, is the length of the longest path from τ to any leaf, i.e.,*

$$h(\tau) = \begin{cases} 0 & \textit{if } \tau = x \\ 1 + \max_{i=1}^{k} h(\sigma_i) & \textit{if } \tau = \prod_{i=1}^{k} \sigma_i \\ 1 + \max_{i=1}^{2} h(\rho_i) & \textit{if } \tau = \rho_1 \to \rho_2 \end{cases} \tag{5.4}$$

Edges in Figure 5.1b were stretched so that nodes of the same height are drawn at the same level. Observe that product-types always have odd heights and function-types always have even heights. This can be easily proved by induction on the product grammar.

Lemma 5.4. Let τ, τ' be two types conforming to the product grammar. Then,

$$\textsf{Product}^{-} \vdash \tau = \tau' \Rightarrow h(\tau) = h(\tau').$$

PROOF. Trivial by structural induction on τ and τ' using Lemma 5.2. □

Theorem 5.5. 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 τ and τ'. We will label each of these n types with an identifier drawn from the range $[1, n]$, such that two types are isomorphic if and only if they have the same identifier.

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. The algorithm starts by passing T_0 to the broad integer partitioning algorithm. A renaming process then yields unique identifiers for all primitive-types.

The processing of T_ι, $\iota \geq 1$ depends on whether ι is even or odd. If ι is even, 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 odd, 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 *plus* the number of *edges* of the graph.

6. The $\mathbf{P}/\mathbf{F}$-graph

To generalize the linear isomorphism algorithm to deal with the first order isomorphism problem, we now introduce the normal form nf_{d} in which the *distributive rule* $\mathcal{R}.7$ is not applicable. As noted in Section 1.3, an exhaustive application of this rule may lead to a representation of exponential size. The $\mathbf{P}/\mathbf{F}$-graph, described in this section, is a linear size representation of the normal form nf_{d}.

Let τ and τ' be two arbitrary types conforming to the product grammar. The problem is to determine whether $\mathsf{First}^{-} \vdash \tau = \tau'$. (The assumption that the inputs conform to the product grammar is safe since the normalizing function nf_{a} can be applied in linear time to flatten all products.)

Repeated applications of rules $\mathcal{R}.6$ and $\mathcal{R}.7$ will bring each of the inputs to the normal form defined by the *first order grammar*:

First order Grammar

$$\varrho ::= \prod_{i=1}^{k} \varsigma \qquad (k \geq 1)$$

$$\varsigma ::= x \quad | \quad \varrho \rightarrow x$$

Comparing the first order grammar and the product grammar we see that the derivation $\sigma ::= \rho \rightarrow \rho$ is replaced by $\varsigma ::= \varrho \rightarrow x$, i.e., all functions must return a primitive-type.

Algorithmically, this normal form can be generated by applying the normalizing function nf_{d}, defined by

$$\mathrm{nf}_{\mathrm{d}}(\tau) = \begin{cases} \prod(x) & \text{if } \tau = x \\ \bowtie_{i=1}^{k} \mathrm{nf}_{\mathrm{d}}(\sigma_i) & \text{if } \tau = \prod_{i=1}^{k} \sigma_i \\ R_{6,7}\big(\mathrm{nf}_{\mathrm{d}}(\rho_1), \rho_2\big) & \text{if } \tau = \rho_1 \rightarrow \rho_2 \end{cases} \tag{6.1}$$

where $R_{6,7}$ is an auxiliary function, mutually recursive with nf_{d}, which handles function types:

$$R_{6,7}(\varrho, \tau) = \begin{cases} \prod(\varrho \rightarrow x) & \text{if } \tau = x \\ \bowtie_{i=1}^{k} R_{6,7}\big(\varrho, \sigma_i\big) & \text{if } \tau = \prod_{i=1}^{k} \sigma_i \qquad \textit{// apply rule } \mathcal{R}.7 \\ R_{6,7}\Big(\big(\varrho \bowtie \mathrm{nf}_{\mathrm{d}}(\rho_1)\big), \rho_2\Big) & \text{if } \tau = \rho_1 \rightarrow \rho_2 \qquad \textit{// apply rule } \mathcal{R}.6 \end{cases} \tag{6.2}$$

Functions nf_{d} and $R_{6,7}$ must *eagerly* evaluate their arguments to ensure that the distributive rule is applied in outer-first order (Remark 1.9). In other words, given a function type $\rho_1 \rightarrow \rho_2$, rule $\mathcal{R}.7$ is first applied to ρ_1 and only then to ρ_2. This is the reason that the call to $R_{6,7}$ in (6.1) cannot commence before $\mathrm{nf}_{\mathrm{d}}(\rho_1)$ finishes.

We shall see that the definition of $R_{6,7}$ gives rise to a multiple-terms version of the distributive transformation (1.8). In this version, an input node $\varrho \rightarrow \prod_{i=1}^{k} \sigma_i$ is converted to $\prod_{i=1}^{k}(\alpha \rightarrow \sigma_i)$ where α is represented as a pointer to the node corresponding to the product ϱ.

We now examine definitions (6.1) and (6.2) more formally. First, we show that the value returned by these functions is isomorphic to their input. Let ϱ be an arbitrary type.

Lemma 6.1.

First $\vdash \tau = \mathrm{nf_d}(\tau)$,

First $\vdash \varrho \to \tau = R_{6,7}(\varrho, \tau)$.

PROOF. We first note that since τ conforms to the product grammar, then exactly one of the three cases in the definition of either $\mathrm{nf_d}$ (6.1) or $R_{6,7}$ (6.2) must apply. The lemma is then proved by *simultaneous* structural induction on τ. The induction base is the first case in both definitions. By examining the second and third cases of (6.1) we see that it immediately follows from the (simultaneous) inductive hypothesis that function $\mathrm{nf_d}$ returns a type isomorphic to τ. The distributive (currying) axiom and the same inductive hypothesis show that $R_{6,7}$ returns a type isomorphic to $\varrho \to \tau$ in the second (third) case of its definition (6.2). $\square$

Lemma 6.2. Type $\mathrm{nf_d}(\tau)$ conforms to the first order grammar. Further, if ϱ also conforms to this grammar, then so does $R_{6,7}(\varrho, \tau)$.

PROOF. Note that all types conforming to this grammar are products whose terms are either primitive or function types. The proof is again carried out by simultaneous induction on the structure of τ. Again, the induction base is trivially given by the first case of (6.1) and (6.2). The induction step is also easy: in the second case of both definition the returned value is simply a product of terms covered by the inductive hypothesis. In the third case of these definitions the returned value is of a recursive call $R_{6,7}(\cdot, \rho_2)$ where $|\rho_2| < |\tau|$. The proof is completed by checking that the first argument in both of these recursive calls conforms to the first order grammar as required for satisfying the inductive hypothesis. $\square$

We stress that $\mathrm{nf_d}(\tau)$ may be of size $O(n^2)$, as indeed happens in example (1.10). The reason for this blowup is in the third case of $R_{6,7}$: the concatenation $\varrho \bowtie \mathrm{nf_d}(\rho_1)$ creates a new product node whose list of terms are the concatenation of two lists of terms: that of ϱ and $\mathrm{nf_d}(\rho_1)$. Note that the terms themselves are not duplicated, but a new list of terms must be created. The reason that we cannot reuse the two existing lists of terms is that ϱ can be shared among independent recursive calls due to the second case of $R_{6,7}$: we have k independent calls of the form $R_{6,7}\big(\varrho, \sigma_i\big)$.

In order to give the linear space and time bounds for the normalization process, we describe a *shared* representation of types in the first order grammar. Instead of the usual expression tree, we shall use a special rooted acyclic graph. We use the term $\mathbf{P}/\mathbf{F}$-graph since the nodes in it are either $\mathbf{P}$-nodes (representing product-types) or $\mathbf{F}$-nodes (representing function-types).

A $\mathbf{P}$-node v has a field $\varphi(v)$ storing the non-empty set of pointers to term nodes. Terms are either $\mathbf{F}$-nodes or primitive-types, which are encoded simply by identifiers in the range $[1, n]$. In addition, v has a field $\texttt{parent}(v)$ pointing to another $\mathbf{P}$-node, from which v inherits additional terms.

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

$\mathbf{P}/\mathbf{F}$-graphs are further restricted by the demand that `parent` edges define a tree over the $\mathbf{P}$-nodes called the *product tree*, and denoted $\mathcal{T}$. The tree $\mathcal{T}$ is rooted at a dummy $\mathbf{P}$-node, denoted $\mathbf{P}_\perp$, which has no terms, i.e., $\varphi(\mathbf{P}_\perp) = \emptyset$. $\mathbf{P}$-nodes are therefore initialized with their `parent` field pointing at $\mathbf{P}_\perp$.

Definition 6.3 (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.,*

$$\phi(v) = \begin{cases} \emptyset & \textit{if } v = \mathbf{P}_\perp \\ \varphi(v) \cup \phi(\texttt{parent}(v)) & \textit{otherwise.} \end{cases}$$

Consider, for example, Figure 6.1a which shows type (5.2) in the product grammar. Figure 6.1b shows the result of applying algorithm `NormalizeProduct` (described later) on this type.

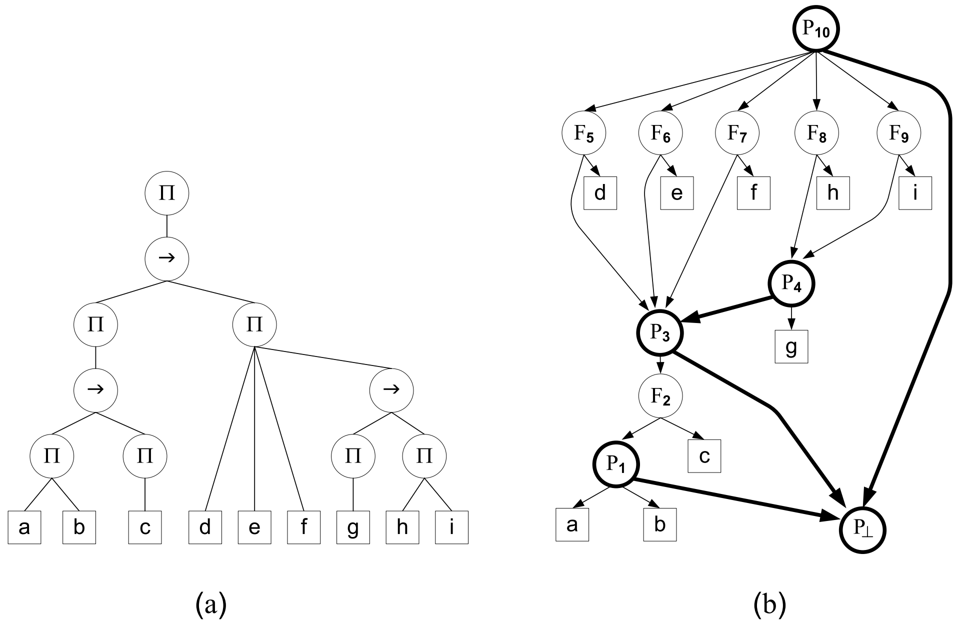


Fig. 6.1. (a) Type (5.2) in the product grammar, and (b) its $\mathbf{P}/\mathbf{F}$-graph representation. The `parent` edges are depicted in bold.

The $\mathbf{P}$-nodes in Figure 6.1b are:

$$\begin{array}{rlcl} \mathbf{P}_\perp = & & & \prod \\ \mathbf{P}_1 = & \mathbf{P}_\perp & \bowtie & \prod(\mathsf{a}, \mathsf{b}) \\ \mathbf{P}_3 = & \mathbf{P}_\perp & \bowtie & \prod(\mathbf{F}_2) \\ \mathbf{P}_4 = & \mathbf{P}_3 & \bowtie & \prod(\mathsf{g}) \\ \mathbf{P}_{10} = & \mathbf{P}_\perp & \bowtie & \prod(\mathbf{F}_5, \mathbf{F}_6, \mathbf{F}_7, \mathbf{F}_8, \mathbf{F}_9) \end{array} \qquad (6.3)$$

We see that each term of a $\mathbf{P}$-node is either a primitive type (e.g., a) or an $\mathbf{F}$-node (e.g., $\mathbf{F}_2$). In addition to the set of terms, each $\mathbf{P}$-node (except $\mathbf{P}_\perp$) inherits additional terms via the `parent` edge. For example, $\texttt{parent}(\mathbf{P}_4) = \mathbf{P}_3$, i.e., $\mathbf{P}_4$ inherits the terms of $\mathbf{P}_3$ which recursively inherits the terms of $\mathbf{P}_\perp$. Therefore, the extended terms of $\mathbf{P}_4$ are the union of the terms of $\mathbf{P}_4$, $\mathbf{P}_3$, and $\mathbf{P}_\perp$:

$$\phi(\mathbf{P}_4) = \varphi(\mathbf{P}_4) \cup \varphi(\mathbf{P}_3) \cup \varphi(\mathbf{P}_\perp).$$

Algorithms 3 and 4 present two mutually recursive routines, namely `NormalizeProduct` and *`FunctionIntoProduct`*. These routines are storage-minded variants of functions $\mathrm{nf_d}$ and $R_{6,7}$, respectively. Together, the two describe a single pass traversal of an abstract syntax tree of a type conforming to the product grammar. The output is a linear sized $\mathbf{P}/\mathbf{F}$-graph of an isomorphic type in the first order grammar.

Algorithm 3 *`NormalizeProduct`*(τ)

Given a type τ conforming to the product grammar, return a $\mathbf{P}$-node v *of an isomorphic type in the first order grammar.*

```
1:  $v \leftarrow$ new $\mathbf{P}$-node // Initially parent$(v) = \mathbf{P}_\perp$, $\varphi(v) = \emptyset$
2:  If $\tau$ is a primitive-type $x$ then
3:      $\varphi(v) \leftarrow \{x\}$
4:  else if $\tau$ is a product-type then
5:      Let $k$ and $\sigma_i$, $i = 1, \ldots, k$, be such that $\tau = \prod_{i=1}^{k} \sigma_i$
6:      For $i = 1, \ldots, k$ do // Normalize all terms in the product
7:          $u_i \leftarrow$ NormalizeProduct$(\sigma_i)$
8:          $\varphi(v) \leftarrow \varphi(v) \cup \varphi(u_i)$ // Collect terms of $u_i$
9:      od
10: else // $\tau$ is a function-type
11:     Let $\rho_1$ and $\rho_2$ be such that $\tau = \rho_1 \to \rho_2$
12:     $u \leftarrow$ NormalizeProduct$(\rho_1)$
13:     $v \leftarrow$ FunctionIntoProduct$(u, \rho_2)$
14: fi
15: Return $v$
```

Lines 2–3 of Algorithm 3 correspond to the first case of function $\mathrm{nf_d}$, lines 4–9 to the second case, and lines 10–14 to the third. The union operation in line 8 correspond to the concatenation operation $\bowtie$ in the second case of $\mathrm{nf_d}$.

Algorithm 4 follows the same outline as function $R_{6,7}$: lines 2–5 correspond to the first case of $R_{6,7}$, lines 6–11 to the second, and lines 12–17 to the third. Again, the union operation in line 10 correspond to the concatenation operation $\bowtie$ in the second case of $R_{6,7}$. However, the concatenation operation $\bowtie$ in the third case of $R_{6,7}$ was translated into an assignment to the `parent` field of w in line 15. This line is the crux of the two routines, making the linear space representation possible.

Let us examine lines 12–17 and the third case of $R_{6,7}$. Node u represents type ϱ, and node w represents the product $\varrho \bowtie \mathrm{nf_d}(\rho_1)$. In line 14, we assign *`NormalizeProduct`*(ρ_1) to w. Then, instead of adding the terms of u to w (i.e., $\varphi(w) \leftarrow \varphi(w) \cup \varphi(u)$) we point the `parent` field of w to u in line 15. Therefore the expanded terms of w are equal to those of the product $\varrho \bowtie \mathrm{nf_d}(\rho_1)$.

The next lemma proves that algorithms 3 and 4 run in $O(n)$ time and space.

Lemma 6.4. Let τ be a type conforming to the product grammar, and let u be a $\mathbf{P}$-node. Then, the function calls *`NormalizeProduct`*(τ) and *`NormalizeProduct`*(u, τ) execute in $O(|\tau|)$ time and space.

Algorithm 4 *FunctionIntoProduct*(u, τ)

Given a $\mathbf{P}$-node u *and a type* τ *(which is a product-type) return* v*, a new* $\mathbf{P}$-node *describing a type isomorphic to the function-type* $\varrho \to \tau$*, where* ϱ *is the type represented by the* $\mathbf{P}$-node u*.*

```
1:  $v \leftarrow$ new $\mathbf{P}$-node // Initially parent$(v) = \mathbf{P}_\perp$, $\varphi(v) = \emptyset$
2:  If $\tau$ is a primitive-type $x$ then
3:      $w \leftarrow$ new $\mathbf{F}$-node
4:      arg$(w) \leftarrow u$; ret$(w) \leftarrow x$ // $w$ represents the type $\varrho \to x$
5:      $\varphi(v) \leftarrow \{w\}$
6:  else if $\tau$ is a product-type then
7:      Let $k$ and $\sigma_i$, $i = 1, \ldots, k$, be such that $\tau = \prod_{i=1}^{k} \sigma_i$
8:      For $i = 1, \ldots, k$ do // Normalize all terms in the product
9:          $u_i \leftarrow$ FunctionIntoProduct$(u, \sigma_i)$
10:         $\varphi(v) \leftarrow \varphi(v) \cup \varphi(u_i)$ // Collect terms of $u_i$
11:     od
12: else // $\tau$ is a function-type
13:     Let $\rho_1$ and $\rho_2$ be such that $\tau = \rho_1 \to \rho_2$
14:     $w \leftarrow$ NormalizeProduct$(\rho_1)$
15:     parent$(w) \leftarrow u$ // Share the common argument $\varrho$
16:     $v \leftarrow$ FunctionIntoProduct$(w, \rho_2)$
17: fi
18: Return $v$
```

PROOF. Proved by mutually-recursive structural-induction on τ. The induction base is when τ is a primitive type. It is mundane to check that lines 2–3 of Algorithm 3 and lines 2–5 of Algorithm 4 execute in constant time and space. In the induction step, τ is either a function or a product. The amount of time and space invested in addition to the recursive calls is either constant if $\tau = \rho_1 \to \rho_2$ or $O(k)$ if $\tau = \prod_{i=1}^{k} \sigma_i$. Note that the union in line 8 of Algorithm 3 and line 10 of Algorithm 4 can be computed in constant time since the terms of u_i are not shared (in contrast to the terms of u which are shared among other calls). □

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

1. Nodes u and v represent the same primitive-type x.
2. Nodes u and v are both $\mathbf{F}$-nodes, `ret`$(u) =$ `ret`(v) and `arg`(u) and `arg`(v) (recursively) represent isomorphic types.
3. Nodes 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. Let τ and τ' be the types u and v represent, respectively. Then, both τ and τ' conform to the first order grammar. Rittri (1990) proved that, in such a case (i.e., when none of the rules $\mathcal{R}.1$–$\mathcal{R}.7$ can be applied), we have that

$$\mathsf{First} \vdash \tau = \tau' \Leftrightarrow \mathsf{Product}^{-} \vdash \tau = \tau'.$$

Deciding the latter can be done using Lemma 5.2. $\Box$

If the terms in $\mathbf{P}$-nodes are expanded, then the size of the representation may increase to $O(n^2)$ (as in (1.10)). 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, thereby improving upon the $O(n^2 \log n)$ best previous result. To obtain a more efficient algorithm, we develop in the next two sections the machinery for comparing unexpanded products.

7. Tree Partitioning

We need to further develop our partitioning algorithms to deal with the *non-expanded* representation of products in the tree of $\mathbf{P}$-nodes rooted at $\mathbf{P}_\perp$. The partitioning of these nodes is tantamount to finding the type isomorphism relationships between $\mathbf{P}$-nodes: Two $\mathbf{P}$-nodes are in the same equivalence class of the partitioning when the *expanded* terms of the respective nodes are the same, which happens if and only if the types these two nodes represent are isomorphic.

To understand this need better, consider again our running example type (5.2)

$$\big((\mathsf{a} \times \mathsf{b}) \to \mathsf{c}\big) \to \Big(\big(\mathsf{d} \times (\mathsf{e} \times \mathsf{f})\big) \times \big(\mathsf{g} \to (\mathsf{h} \times \mathsf{i})\big)\Big).$$

Algorithm `NormalizeProduct` generated the $\mathbf{P}/\mathbf{F}$-graph representation of this type. This representation is depicted again in Figure 7.1a below.

By definition, removing all $\mathbf{F}$-nodes and the edges incident on them from a $\mathbf{P}/\mathbf{F}$-graph will result in a tree. Figure 7.1b shows the tree thus obtained from Figure 7.1a. As explained above, the extended terms of each $\mathbf{P}$-node are computed by inheriting the extended terms of its parent (see Definition 6.3). For example, tree node $\mathbf{P}_4$ in the figure inherits the terms of tree node $\mathbf{P}_3$.

Let us ignore the $\mathbf{F}$-nodes for now, and concentrate on a variant of the multi-set partitioning problem in which the multi-sets are defined by an inheritance tree. We will first develop an algorithm for this variant. Still, we note that this algorithm does not completely solve the general problem of sorting the nodes of a $\mathbf{P}/\mathbf{F}$-graph into equivalence classes. The reason is that the terms in the product-tree are not always known in advance. In Figure 7.1b we see for example that the term $\mathbf{F}_6$ in $\mathbf{P}_{10}$ is not available upfront. We need to process node $\mathbf{P}_3$ before we can be certain that this term is not isomorphic to, for example, term $\mathbf{F}_8$, which in turn depend upon $\mathbf{P}_4$. The next section will take care of this subtlety by developing an incremental algorithm for the problem.

In this section, our concern lies with the simpler, non-incremental, setting, described as follows: Given is 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.,

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

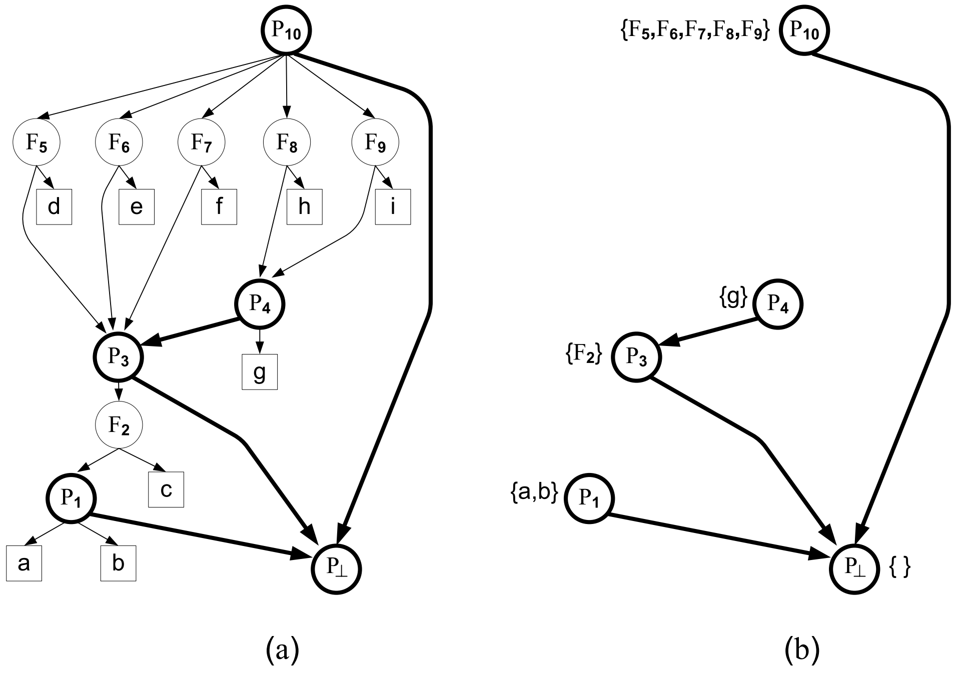


Fig. 7.1. (a) $\mathbf{P}/\mathbf{F}$-graph representation in Figure 6.1b, and (b) its product-tree with the multi-set of terms of each product.

These expanded multi-sets will be in our applications the expanded terms (Definition 6.3) of $\mathbf{P}$-nodes.

Definition 7.1 (Tree partitioning). *Given a tree $\mathcal{T}$, the* tree partitioning *is the partitioning defined by the multi-set partitioning of the expanded multi-sets $\{\phi(v) \mid v \in \mathcal{T}\}$.*

Let M denote the total number of elements in multi-sets of $\mathcal{T}$, i.e., $M = \sum_{v\in\mathcal{T}} |\varphi(v)|$. 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.)

Figure 7.2a 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 $M = 9$.

We have for nodes E and F, for instance,

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

Figure 7.2b depicts the solution of the tree partitioning problem for the multi-set tree of Figure 7.2a. We see that there are 5 partitions:

$$\{\mathsf{A}\}, \{\mathsf{H}\}, \{\mathsf{B}, \mathsf{C}\}, \{\mathsf{D}, \mathsf{E}, \mathsf{G}\}, \{\mathsf{F}\}. \tag{7.1}$$

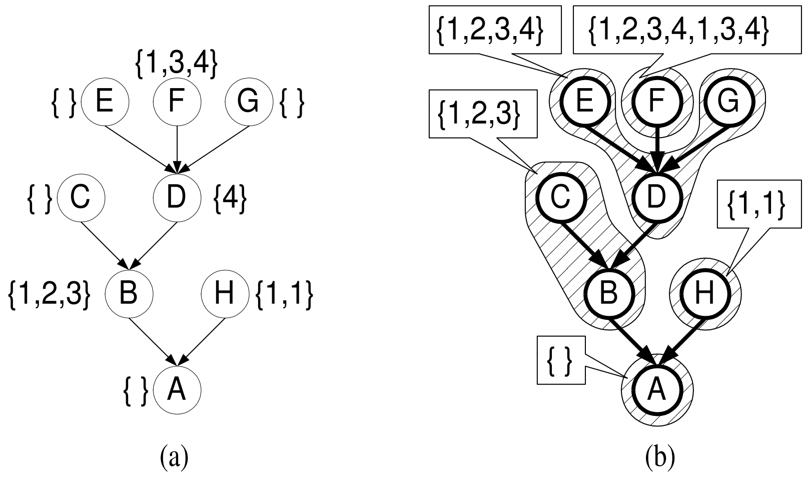


Fig. 7.2. A small multi-set tree (a) and its tree partitioning (b)

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 *naïve solution* to the tree partitioning problem is by directly computing the expanded multi-sets $\phi(v)$. In order to do so, we represent an expanded multi-set $\phi(v)$ as an integer array $\mathbf{Count}_v[1, \ldots, m]$.

Definition 7.2. Given an expanded multi-set $\phi(v)$, its *array-representation*, denoted $\mathbf{Count}_v$, is an array over the indices $[1, \ldots, m]$, 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 size of those n arrays is nm cells, while the time required for computing them is $O(nm + M)$ time since we also examined all the terms $\varphi(v)$. To conclude, the runtime of the naïve solution is $O(nm + M)$ while using $O(nm)$ space.

We now present an algorithm for finding the tree partitioning whose runtime is $O(M \log m)$ using $O(M)$ space. This algorithm 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. (To simplify the complexity analysis we assume that $n \leq M$. This assumption is true in our application since $\mathbf{P}$-nodes have a non-empty set of terms, i.e., $|\varphi(u)| \geq 1$.)

Definition 7.3. A *family* F_i, $i = 1, \ldots, m$, is a multi-set of nodes such that if i occurs k times in $\varphi(v)$, then v occurs k times in F_i.

In our example, four such families are defined:

$$\begin{aligned} F_1 &= \{\mathsf{B},\mathsf{F},\mathsf{H},\mathsf{H}\},\\ F_2 &= \{\mathsf{B}\},\\ F_3 &= \{\mathsf{B},\mathsf{F}\},\\ F_4 &= \{\mathsf{D},\mathsf{F}\}. \end{aligned} \tag{7.2}$$

Note that $\sum_{i=1}^{m} |F_i| = M$.

Given a tree $\mathcal{T}$ and a multi-set F of its nodes, it is easy to define a partitioning of the nodes of $\mathcal{T}$ where the classification criterion is the number of occurrences of a node in F. We shall however be interested in a more sophisticated such partitioning, denoted ∇F, in which the classification criterion is the number of times a node "inherits" membership in F. More precisely,

Definition 7.4. Let u, v be two nodes of $\mathcal{T}$, and let $\mathrm{ancestors}(u)$ (respectively, $\mathrm{ancestors}(v)$) be the set of ancestors of u. Then, u and v are in the same partition of ∇F if and only if

$$|\mathrm{ancestors}(u) \cap F| = |\mathrm{ancestors}(v) \cap F|.$$

In our example, the four family partitionings induced by the families of (7.2) are:

$$\begin{aligned} \nabla F_1 &= \{\{\mathsf{A}\},\{\mathsf{F},\mathsf{H}\},\{\mathsf{B},\mathsf{C},\mathsf{D},\mathsf{E},\mathsf{G}\}\},\\ \nabla F_2 &= \{\{\mathsf{A},\mathsf{H}\},\{\mathsf{B},\mathsf{C},\mathsf{D},\mathsf{E},\mathsf{F},\mathsf{G}\}\},\\ \nabla F_3 &= \{\{\mathsf{A},\mathsf{H}\},\{\mathsf{F}\},\{\mathsf{B},\mathsf{C},\mathsf{D},\mathsf{E},\mathsf{G}\}\},\\ \nabla F_4 &= \{\{\mathsf{A},\mathsf{B},\mathsf{C},\mathsf{H}\},\{\mathsf{F}\},\{\mathsf{D},\mathsf{E},\mathsf{G}\}\}. \end{aligned} \tag{7.3}$$

Note that all the nodes in a certain partition of ∇F_i, $1 \le i \le 4$, have the same number of occurrences of i. For example, $\mathbf{Count}_{\mathsf{F}}[1] = \mathbf{Count}_{\mathsf{H}}[1] = 2$. In fact, it is easy to prove the following:

Lemma 7.5. Let F_i be a family, and v be a node of $\mathcal{T}$, then

$$|\mathrm{ancestors}(v) \cap F_i| = \mathbf{Count}_v[i].$$

The performance gain of the dual representation is due to the fact that the multi-set of nodes in which a value participates is often a subtree of $\mathcal{T}$. For example, the partition $\{\mathsf{B},\mathsf{C},\mathsf{D},\mathsf{E},\mathsf{F},\mathsf{G}\}$ of ∇F_2 is a subtree rooted at B.

Next we define the *intersection* of two partitionings P_1 and P_2, written as $P_1 \times P_2$, and show that $\nabla F_1 \times \cdots \times \nabla F_m$ is in fact the tree partitioning.

Definition 7.6. Let P_1 and P_2 be two partitionings. Then, their *intersection*, denoted $P_1 \times P_2$, is defined by

$$P_1 \times P_2 = \{p_1 \cap p_2 \mid p_1 \in P_1, p_2 \in P_2\}.$$

In other words $P_1 \times P_2$ is obtained by intersecting each partition of P_1 with each partition of P_2. For example, the intersection of ∇F_1 and ∇F_2 is

$$\begin{aligned} \nabla F_1 \times \nabla F_2 &= \{\{\mathsf{A}\},\{\mathsf{F},\mathsf{H}\},\{\mathsf{B},\mathsf{C},\mathsf{D},\mathsf{E},\mathsf{G}\}\} \times \{\{\mathsf{A},\mathsf{H}\},\{\mathsf{B},\mathsf{C},\mathsf{D},\mathsf{E},\mathsf{F},\mathsf{G}\}\}\\ &= \{\{\mathsf{A}\},\{\mathsf{H}\},\{\mathsf{F}\},\{\mathsf{B},\mathsf{C},\mathsf{D},\mathsf{E},\mathsf{G}\}\}. \end{aligned}$$

It is mundane to see that $\times$ is commutative and associative.

Lemma 7.7. The partitioning $\nabla F_1 \times \cdots \times \nabla F_m$ is the tree partitioning.

PROOF. Let P be the tree partitioning. Let $u, v \in \mathcal{T}$ be arbitrary. For a partitioning X, we write $u \equiv v \bmod X$ to denote that u, v belong to the same partition of X. Then we need to prove that $u \equiv v \bmod P$ if and only if

$$u \equiv v \bmod \nabla F_1 \times \cdots \times \nabla F_m.$$

Suppose first that

$$u \equiv v \bmod P. \tag{7.4}$$

Then, from the definition of the tree partitioning (Definition 7.1) we have that

$$\phi(u) = \phi(v). \tag{7.5}$$

It follows by the definition of the array-representation $\mathbf{Count}[1, \ldots, m]$ (Definition 7.2) that

$$\forall 1 \leq i \leq m \bullet \mathbf{Count}_u[i] = \mathbf{Count}_v[i]. \tag{7.6}$$

If $\mathbf{Count}_u[i] = \mathbf{Count}_v[i]$ then, by Lemma 7.5, $|\text{ancestors}(u) \cap F_i| = |\text{ancestors}(v) \cap F_i|$, so we may write

$$\forall 1 \leq i \leq m \bullet |\text{ancestors}(u) \cap F_i| = |\text{ancestors}(v) \cap F_i|. \tag{7.7}$$

From the definition of the ∇ operator (Definition 7.4) we have that

$$\forall 1 \leq i \leq m \bullet u \equiv v \bmod \nabla F_i. \tag{7.8}$$

Finally, from the definition of the intersection of two partitionings (Definition 7.6)

$$u \equiv v \bmod \nabla F_1 \times \cdots \times \nabla F_m. \tag{7.9}$$

To show that (7.4) follows from (7.9) we trivially follow the above reasoning chain in the reverse direction. □

We now devise an efficient representation of family partitionings and a way to compute their intersection. To this end, we describe below the *segmented-array* representation of a family partitioning ∇F which requires $O(|F|)$ space. We also show how to intersect two segmented-arrays A_1 and A_2, which results in another segmented-array A_3 which represents $A_1 \times A_2$ where

$$|A_3| \leq |A_1| + |A_2|.$$

The trick is to consider a pre-order traversal of the tree, in which subtrees can be simply encoded as intervals. Therefore, members of a family F define intervals, which in turn break the pre-order into segments. Thus, the partitioning ∇F can be encoded as an array mapping those segments to their containing partition.

In our example, let the pre-order traversal be

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

As can be seen in Figure 7.3, the descendants of any given node form an interval. This figure highlights the intervals of the descendants of nodes B and F:

$$\text{descendants}(\mathsf{B}) = \{\mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}, \mathsf{F}, \mathsf{G}\} = [\mathsf{B}, \mathsf{G}],$$
$$\text{descendants}(\mathsf{F}) = \{\mathsf{F}\} = [\mathsf{F}, \mathsf{F}].$$

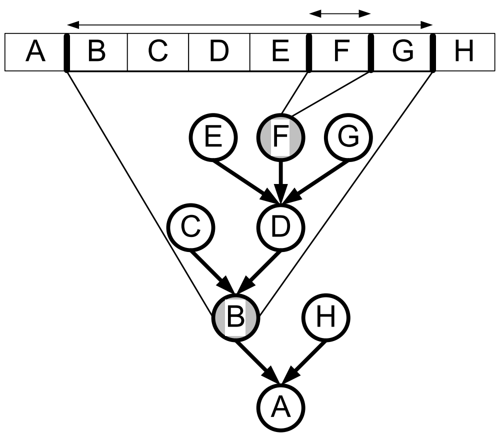


Fig. 7.3. The intervals and segments defined by family $F_3 = \{\mathsf{B}, \mathsf{F}\}$

Consider now the family F_3 defined by these two nodes, $F_3 = \{\mathsf{B}, \mathsf{F}\}$. In Figure 7.3 we see that the two intervals of F_3,

$$\mathrm{Intervals}(F_3) = \{[\mathsf{B}, \mathsf{G}], [\mathsf{F}, \mathsf{F}]\},$$

break π into five segments

$$\mathrm{Segments}(F_3) = \{[\mathsf{A}, \mathsf{A}], [\mathsf{B}, \mathsf{E}], [\mathsf{F}, \mathsf{F}], [\mathsf{G}, \mathsf{G}], [\mathsf{H}, \mathsf{H}]\}.$$

Consider any arbitrary such segment defined by F_3, and let v range over the nodes of this segment. Then, the multiplicity of the value 3 in $\phi(v)$ is the same, e.g., the multiplicity of the value 3 in the segment $[\mathsf{B}, \mathsf{E}]$ is 1. The *segmented-array* representation associates a multiplicity to each segment. This multiplicity is called the *segment descriptor*. The segmented-array of family F_3 is therefore

$$\mathrm{SegmentedArray}(F_3) = \langle [\mathsf{A}, \mathsf{A}] \mapsto 0, [\mathsf{B}, \mathsf{E}] \mapsto 1, [\mathsf{F}, \mathsf{F}] \mapsto 2, [\mathsf{G}, \mathsf{G}] \mapsto 1, [\mathsf{H}, \mathsf{H}] \mapsto 0\rangle,$$

and its family partitioning is

$$\nabla F_3 = \{\{\mathsf{A}, \mathsf{H}\}, \{\mathsf{F}\}, \{\mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}, \mathsf{G}\}\}.$$

Observe that each segment is contained in some partition of ∇F_3, and that two segments with the same descriptor belong to the same partition. For example, both segments $[\mathsf{B}, \mathsf{E}]$ and $[\mathsf{G}, \mathsf{G}]$ are contained in the partition $\{\mathsf{B}, \mathsf{C}, \mathsf{D}, \mathsf{E}, \mathsf{G}\}$ of ∇F_i. In fact, the union of those two segments is exactly this partition. It is easy to check that this is no coincidence, i.e., the union of segments with the same descriptor is equal to some partition in ∇F_3, and vice versa.

More formally,

Definition 7.8. Let P be a partitioning of the nodes of $\mathcal{T}$, and let π be a pre-order traversal of $\mathcal{T}$. Then, a *segmented-array* representation of P is an array of segment records, each record containing the segment starting and ending indices and a descriptor such that:

1. The segments are distinct and cover π, i.e., the segments are a partitioning of π.
2. Each segment is contained in some partition of P. In other words, the segmented array represents a finer-grained partitioning than P.

3. Two segments have the same descriptor if and only if they are contained in the same partition of P.
4. The segments are sorted in an increasing order.

We will sometimes refer to a family partitioning ∇F as a segmented-array. No confusion will arise.

A segmented-array representation of a family partitioning ∇F can be created in $O(|F|)$ time and space since the number of segments is linear in $|F|$. More precisely, a family F defines at most $|F|$ distinct intervals in π, one for each distinct node in F. These intervals break π into at most $2|F| + 1$ segments.

Figure 7.4 depicts the segmented-array representations of the family partitionings of (7.3).

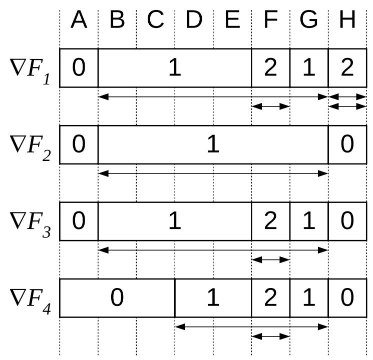


Fig. 7.4. The segmented-arrays of the families of Figure 7.2a

The *intersection* of two segmented-arrays P_1 and P_2, 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 array of size at most $s_1 + s_2$. The descriptors of the segments in $P_1 \times P_2$ are the *renamed* pairs of descriptors of the originating segments from P_1 and P_2 (using Lemma 4.7).

Figure 7.5 depicts the intersection of the segmented-arrays of ∇F_1 and ∇F_2 from (7.2).

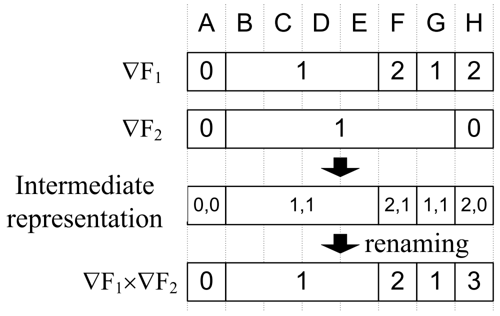


Fig. 7.5. Computing the intersection of the two segmented-arrays ∇F_1 and ∇F_2 defined by Figure 7.2a.

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

We are now ready to state the principal result of this section describing the (non-incremental) tree partitioning algorithm and its performance.

Theorem 7.9. There is an $O(M \log m)$ time and $O(M)$ space algorithm solving the tree partitioning problem.

PROOF. Using Lemma 7.7, we wish to compute $\nabla F_1 \times \cdots \times \nabla F_m$. We therefore build a balanced binary tree whose leaves are the segmented-arrays $\nabla F_1, \ldots, \nabla F_m$. In each internal node we compute the intersection of the two segmented-arrays of its two children. The segmented-array at the root of this tree represents the tree partitioning.

Consider the first level of this tree which contains the segmented-arrays $\nabla F_1, \ldots, \nabla F_m$. Recall that the size of the segmented-array ∇F_i is $2|F_i| + 1$. Therefore, the size of the entire first level is

$$\sum_{i=1}^{m}(2|F_i| + 1) = O(M).$$

In calculating the second level of the tree, we intersect pairs of segmented-arrays $\nabla F_i \times \nabla F_{i+1}$, for odd values of i. Recall also that the time (and space) for creating $\nabla F_i \times \nabla F_{i+1}$ is $O(|F_i| + |F_{i+1}|)$. Thus, the time (and space) for creating the second level is again $O(M)$.

In general, since all the segmented-arrays propagate to the root, we have that the total size of all segmented-arrays at each tree level, and thus the work to generate the next level, is $O(M)$. Since the number of levels is $\lceil \log_2 m\rceil + 1$, we have that the total time for computing $\nabla F_1 \times \cdots \times \nabla F_m$ is $O(M \log m)$. $\square$

For an example, refer to Figure 7.6 which depicts the balanced binary tree of the families of (7.2). We see in the figure that the segmented-array 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 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 $\{$D, E, G$\}$. We have thus obtained the desired partitioning (7.1) of the tree in Figure 7.2a.

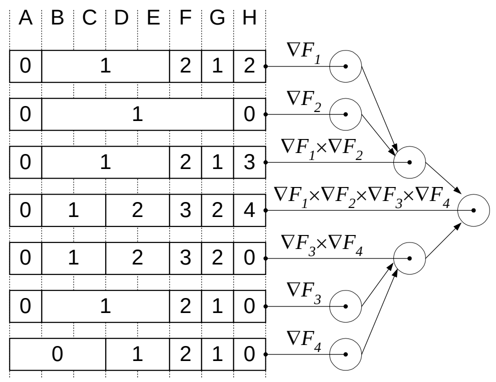


Fig. 7.6. The balanced binary tree of the families of Figure 7.2

8. Incremental Tree Partitioning

The tree partitioning problem (Definition 7.1) 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 Theorem 7.9 however assumes that all multi-sets members are directly comparable. It is applicable only in the case when all terms are primitive-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.

Definition 8.1. Given a tree $\mathcal{T}$, an *incremental tree partitioning data structure* must support two kinds of operations, which might be interleaved:

1. Operation `insert`(F_j), where F_j is a family, i.e., a multi-set of nodes of $\mathcal{T}$.
2. Query `classify`(T_k), where T_k is a subset of the nodes of $\mathcal{T}$. This query returns the tree partitioning of T_k according to the families inserted so far. More formally, let $\{F_1, \ldots, F_j\}$ be the set of families inserted so far. Then, the query returns the restriction of $\nabla F_1 \times \cdots \times \nabla F_j$ to the set T_k. This restriction is defined in the obvious manner, i.e., it is the partitioning obtained by intersecting each partition of $\nabla F_1 \times \cdots \times \nabla F_j$ with T_k, and ignoring all thusly obtained empty partitions.

To make the complexity analysis easier, we assume that the sets $\{T_k\}$ are disjoint, that $\bigcup_k T_k = \mathcal{T}$ and that the data structure is never required to classify a node before its parent.

These assumptions hold in our application: the set of nodes T_k is exactly the set of $\mathbf{P}$-nodes whose height is $2i$, and a family 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.)

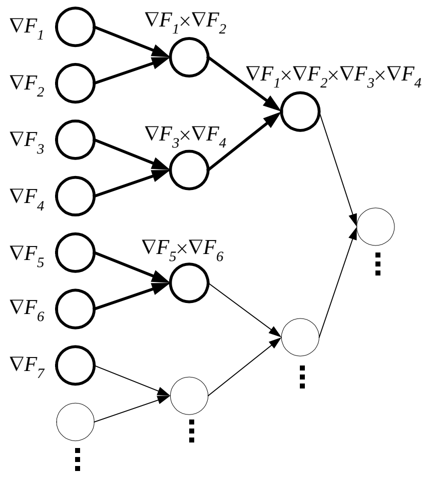
Our main objective is to minimize the resources for processing the entire interleaved sequence of data structure operations. The next theorem states the performance characteristics of our incremental tree partitioning algorithm.

Theorem 8.2. Incremental tree partitioning can be solved in $O(M \log m + n \log n \log m)$ time and $O(M)$ space.

PROOF. We use a lazy representation of an infinite complete binary tree, similar to the binary tree of Theorem 7.9, The leaves of this tree are given by the infinite sequence $\nabla F_1, \nabla F_2, \ldots$

Figure 8.1 shows (part of) this tree, after families $\nabla F_1, \ldots, \nabla F_7$ have been inserted.

This infinite tree is used to guide the computation of the intersection of the partitioning which



∇F_1
∇F_2
∇F_3
∇F_4
∇F_5
∇F_6
∇F_7
$\nabla F_1 \times \nabla F_2$
$\nabla F_3 \times \nabla F_4$
$\nabla F_5 \times \nabla F_6$
$\nabla F_1 \times \nabla F_2 \times \nabla F_3 \times \nabla F_4$

Fig. 8.1. An embedding of seven families into an infinite balanced binary tree

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 segmented-array ∇F_8, and proceed to compute the following three intersections:

$$P_1 = \nabla F_8 \times \nabla F_7,$$
$$P_2 = P_1 \times (\nabla F_5 \times \nabla F_6),$$
$$P_3 = P_2 \times (\nabla F_1 \times \nabla F_2 \times \nabla F_3 \times \nabla F_4).$$

After this insertion we will have a single temporary root.

The total time for all insert operations, i.e., $\texttt{insert}(F_1), \ldots, \texttt{insert}(F_m)$, is the same as in the non-incremental tree partitioning problem, i.e., $O(M \log m)$ time using $O(M)$ space.

The algorithm is lazy in the sense that we do not compute the intersection of the temporary roots $P_1, \ldots, P_r$. Instead, the classification of a set T_k, i.e., $\texttt{classify}(T_k)$ query, is carried out by consulting the segmented-arrays at those temporary roots. Recall that P_i is represented as a sorted array of segment-identifier pairs (see Definition 7.8). Since the size of this array is bounded by n, we can support searches in P_i in $O(\log n)$ time. For each $v \in T_k$, we search for the descriptor of the segment which contains v, in P_i for $i = 1, \ldots, r$.

After obtaining an r-tuple of descriptors for all $v \in T_k$, we apply a tuple partitioning algorithm to classify T_k. In order to keep the space linear, we cannot actually store $|T_k|$ tuples of length r. Therefore, we will use the *incremental tuple partitioning algorithm*. Specifically, we will use $|T_k|$ 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.

Note that after j families were inserted, there are at most $\lceil \log_2 j \rceil$ temporary roots, so we always have that $r \leq \lceil \log_2 m \rceil$. Thus, the total time for computing the r-tuple is $O(r \log n) \subseteq O(\log m \log n)$. The total time for the $\texttt{classify}(T_k)$ query is therefore $O(|T_k| \log m \log n)$, while using $O(M)$ space. Since every node $v \in \mathcal{T}$ can take part in a classification query at most once, the total time for all classifications is $O(n \log n \log m)$.

The total time for all insertions and classifications is $O(M \log m + n \log n \log m)$, while the total space used is $O(M)$. $\square$

9. An Algorithm for the First Order Isomorphism Problem

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 described in this chapter: an efficient algorithm for deciding first order isomorphisms. In essence, the algorithm uses Lemma 6.5. 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 Definition 5.3. Algorithm 5 shows how heights can be computed in linear time even in the non-expanded, $\mathbf{P}/\mathbf{F}$ representation.

Algorithm 5 *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:  fi
4:  mark $v$ as visited
5:  If $v$ is a primitive-type or $v = \mathbf{P}_\perp$ then
6:      $h(v) \leftarrow 0$; return $h(v)$ // Recursion base
7:  fi
8:  If $v$ is an $\mathbf{F}$-node then
9:      $h(v) \leftarrow 1 + Height(\texttt{arg}(v))$; return $h(v)$
10: fi
    // $v$ must be an ordinary $\mathbf{P}$-node
11: $h(v) \leftarrow Height(\texttt{parent}(v))$
12: For all $u \in \varphi(v)$ do // recurse on all (non-expanded) terms
13:     $h(v) \leftarrow \max(h(v), 1 + Height(u))$
14: od
15: 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 8–9 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 $\mathbf{P}_\perp$. Since there are no terms in this product-node, its height is 0. Lines 11–15 deal with ordinary $\mathbf{P}$-nodes. The height of such nodes is one more than the maximum height of all expanded terms. The reason why in line 11 we do not add 1 to *`Height`* (`parent`(v)) is that the expanded terms include the terms $\phi($`parent`$(v))$, and not `parent`(v) as a term.

Once the height of all nodes in $\mathbf{P}/\mathbf{F}$-graph is computed, Algorithm 6 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 primitive-types. To process non-primitive-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 6 *`NodesPartitioning`* (G)

Given a $\mathbf{P}/\mathbf{F}$-graph G representing a type in the first order grammar, 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 $\Upsilon$ be an incremental tree partitioning data-structure for the tree of $\mathbf{P}$-nodes
    of $G$
2:  $j \leftarrow 0$ // The identifier of current isomorphism class
3:  Let $r$ be the root of $G$
4:  $l \leftarrow$ Height $(r)$
5:  For $\iota = 1, \ldots, l$ do // Process the nodes by height
6:      Let $T_\iota \leftarrow \{v \in G \mid h(v) = \iota\}$
7:      If $\iota$ is odd then // $T_\iota$ is a collection of $\mathbf{P}$-nodes
8:          $\Lambda \leftarrow \Lambda \cup \Upsilon$.classify$(T_\iota)$
9:      else // $T_\iota$ is a collection of $\mathbf{F}$-nodes
10:         Partition $T_\iota$ using pair partitioning
11:         Let the resulting partition be $T_\iota = C_1 \cup \cdots \cup C_k$
12:         $\Lambda \leftarrow \Lambda \cup \{C_1, \ldots, C_k\}$
            // Update $\Upsilon$
13:         For $i = 1, \ldots, k$ do // Inserting a new family
14:             $j \leftarrow j + 1$ // Process a new isomorphism class $j$
15:             Let $F_j$ be the multi-set of $\mathbf{P}$-nodes with a term in $C_i$
16:             $\Upsilon$.insert$(F_j)$
17:         od
18:     fi
19: od
20: Return $\Lambda$
```

The main data-structure used by the algorithm is incremental tree partitioning (see Theorem 8.2). Nodes at odd height are $\mathbf{P}$-nodes. The classification of these nodes is carried out by querying this data-structure.

Lines 10–17 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 9.1. If G has n nodes and $O(n)$ edges then, Algorithm 6 runs in $O(n \log^2 n)$ time and while consuming $O(n)$ space.

PROOF. We first note that computing the height as in Algorithm 5 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 pair partitioning is $O(n)$ (see Lemma 4.7).

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. Therefore, using Theorem 8.2, the total time of all the operations performed on Υ is

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

while using $O(M)$ space, where x is the number of nodes in the product-tree (which is the number of $\mathbf{P}$-nodes), m is the number of families, and M is the total size of those families. Since all the above parameters are $O(n)$, the total runtime is $O(n \log^2 n)$ using $O(n)$ space. $\square$

The bottom-up node classification of Algorithm 6 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.) Algorithm 6 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 9.2. 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 9.1. $\square$

10. Open Problems

The only lower bound for the first order type isomorphism problem is the trivial information theoretic linear time. An important research direction is to bridge this gap by either *reducing the time complexity* of our main algorithm even further, or obtaining better *lower bounds*.

For example, *dynamic fractional cascading* (Melhorn and Näher, 1990) might be used to decrease the running time from $O(n \log^2 n)$ to $O(n \log n \log \log n)$. Recall that in the incremental tree partitioning algorithm (Section 8) a `classify` query was implemented by conducting *independent* logarithmic time searches in $O(\log n)$ temporary roots. The fractional cascading data structure makes it possible to use the result of each search in expediting the subsequent search, bringing down the runtime of `classify`(T_k) to $O(|T_k| \log n \log\log n)$. Unfortunately, this representation makes it difficult to use the incremental tuple partitioning algorithm, and increases the space to $O(n \log n)$.

Time complexity might be improved also by taking the perspective in which primitive types are thought of as variables, while compound types are considered expressions over these. Then, it follows from the fact that axioms $\mathcal{A}.1$–$\mathcal{A}.7$ are complete (Bruce et al., 1991) that the first order isomorphism problem is reduced to function identity. This identity might in turn be checked by an appropriate random assignment to the variables, possibly leading to a more time efficient, yet *randomized* algorithm for the problem. For example, if infinite precision arithmetic is allowed, then, it might be possible to extend the type isomorphism heuristics of Katzenelson, Pinter and Schenfeld (1992), and check identity by assigning into the variables values drawn at random from, say, the range $[0, 1]$. We note however that such a randomized algorithm does not yield the *isomorphism proof* as does our deterministic algorithm.

Another interesting direction comes from the generalization in which type expression trees may share nodes, i.e., the input is *directed acyclic graph* rather than a tree. This situation occurs naturally in programming languages in which non-primitive types can be named, and where these names can be used in the definition of more complex types.

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.

References

Andreev, A. and Soloviev, S. (1997). A deciding algorithm for linear isomorphism of types with complexity $O(n \log^2(n))$. *Category Theory and Computer Science*, 1290:197–209.

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

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

Barthe, G. and Pons, O. (2001). Type isomorphisms and proof reuse in dependent type theory. In *Proc. of FOSSACS'01*, volume 2030, pages 57–71.

Basin, D. A. (1990). 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.

Bruce, K. B., Di Cosmo, R., and Longo, G. (1991). Provable isomorphisms of types. *Mathematical Structures in Computer Science*, 1:1–20.

Bruce, K. B. and Longo, G. (1985). 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.

Cai, J. and Paige, R. (1995). Using multiset discrimination to solve language processing problems without hashing. *Theoretical Computer Science*, 145(1–2):189–228.

Considine, J. (2000). Deciding isomorphisms of simple types in polynomial time. Technical report, CS Department, Boston University.

Cousot, P. and Cousot, R. (1992). Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179.

Di Cosmo, R. (1992). 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.

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

Fiore, M., Di Cosmo, R., and Balat, V. (2002). 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)*.

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

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

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

Jha, S., Palsberg, J., and Zhao, T. (2002a). Efficient type matching. In *Proc. of the 5th Foundations of Software Science and Computation Structures*.

Jha, S., Palsberg, J., Zhao, T., and Henglein, F. (2002b). Efficient type matching. TOPPS technical report, DIKU, University of Copenhagen, Universitetsparken 1. Submitted to Special issue of Higher-Order Symbolic Computation in memoriam Robert Paige.

Katzenelson, J., Pinter, S. S., and Schenfeld, E. (1992). Type matching, type-graphs, and the schanuel conjecture. *ACM Trans. Prog. Lang. Syst.*, 14(4):574–588.

Melhorn, K. and Näher, S. (1990). Dynamic fractional cascading. *Algorithmica*, 5:215–241.

Paige, R. (1994). Efficient translation of external input in a dynamically typed language. In Pehrson, B. and Simon, I., editors, *Technology and Foundations—Information Processing 94*, volume 1, pages 603–608. North-Holland.

Palsberg, J. and Zhao, T. (2000). Efficient and flexible matching of recursive types. Manuscript.

Rittri, M. (1990). Retrieving library identifiers via equational matching of types. In *10th International Conference on Automated Deduction*, number 449 in Lecture Notes in Computer Science, pages 603–617. Springer Verlag.

Rittri, M. (1991). Using types as search keys in function libraries. *Journal of Functional Programming*, 1:71–89.

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

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

Zibin, Y., Gil, J. Y., and Considine, J. (2003). Efficient algorithms for isomorphisms of simple types. In *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'03)*, pages 160–171. ACM Press.