Fast Structural Type Equivalence and Inclusion Testing
or
Debunking the "Expensive Structural Types" Myth
or
Unimaginative Comparative Language Texts Considered Harmful

(DRAFT: To become an LCS or AI Technical Memo)

Michael R. Blair† [ziggy@lcs.mit.edu]

†MIT Laboratory for Computer Science

Abstract

Structural type equivalence is the notion that two type expressions are equivalent iff they describe equivalently structured data. Name equivalence, by contrast, considers two types equivalent only when they are defined by the same textual occurrence of a type expression. Consequently, structural equivalence is more flexible than name equivalence. Unfortunately, it is commonly considered to be more expensive to support structural equivalence in the type checking phase of statically typed programmming languages. I contend that this reputation is undeserved.

Specifically, I demonstrate that by viewing the structural type equivalence problem on two types as a search problem over the space of the types' normalization lattices (defined below), a gradual deepening algorithm can be used to make the time cost of proving equivalence rival that of name equivalence systems precisely in those situations where a name equivalence discipline is adhered to by the programmer. More importantly, in those situations where a programmer takes advantage of the additional flexibility afforded by the more tolerant structural equivalence, the cost of proving equivalence will still be proportional only to those structural equivalence axioms actually needed to demonstrate equivalence. Thus, the cost of proving the equivalence of two type expressions can be made proportional to the cost of the axioms actually employed rather than to the potential expense of exploiting the overall flexibility of the entire equivalence/inclusion axiom system supported. That is, structural equivalence can be made economically viable.

This approach naturally leads to an efficient solution to the structural inclusion problem as well. It has been argued that structural inclusion is at the heart of the semantics of statically typed object-oriented programming languages with multiple-inheritance, so efficient structural inclusion testing is crucial to the compile-time efficiency of such languages.

This research was supported in part by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract number N00014-83-K-0125.

1 Introduction

In languages that support strong compile-time type checking, a notion of type equivalence is required to determine whether two type expressions denote the "same" type. This is used, for example, to verify the safety of assigning a value of a given type to a variable of a given type or the safety of passing an expression as argument to a procedure.

Languages like PASCAL [Wirth 71] and ADA [Barnes 80] have, to varying degrees, chosen to consider two type expressions equivalent only if they were defined by the same type definition. This strategy is known as *name equivalence*. Subtypes in such languages are likewise based primarily on explicitly declared name-based inclusion relations. This, for example, is the approach taken in SMALLTALK-80 [XLRG 81] and in the C MM N LISP Object System (CL S) [X3J13 88].

A competing, more permissive strategy was chosen for languages such as ALG L 68 [van Wijngaarden *et al.* 76] and AMBER [Cardelli 84/85/86]. In these languages, two type expressions that describe values with the same representation structure are considered to be equivalent. For example, the order of fields in `RECORDOF` types is irrelevant. Such *structural equivalence* typing systems usually present a collection of equivalence axioms over which to deduce structural equivalence. Structural type equivalence and inclusion has been shown to be a natural way to extend strongly typed programming languages with the ability to support a notion of Object-Oriented Programming Style (OOPS) type inheritance [Cardelli 84/88] [Snyder 86].[1]

Unfortunately, several texts in comparative language design (e.g., [MacLennan 87] [Pratt 84] [Sebesta 89]) have bemoaned structural type equivalence as substantially more expensive to support in a compiler for a statically typed programming language than name equivalence. Indeed, some have suggested that this is the primary reason why most main-stream languages today adopt name equivalence rather than structural equivalence. I contend that this reputation is undeserved.

[1] See [Cook, Hill & Canning 90] for why this is only "a notion".

This paper is organized as follows: In Section 2, I clarify the distinctions between name and structural equivalence of types and explain the desirability of structural equivalence and inclusion. In Section 3, I present a small yet powerful set of example equivalence and inclusion axioms that a type checker might incorporate. In Section 4, I introduce the normalization and inclusion lattices induced by these axioms. In Section 5, the problems of structural equivalence and inclusion are recast as a search problem over these induced lattices. This leads to a gradual deepening algorithm that efficiently solves these problems. In Section 6, I briefly summarize our approach and briefly address pragmatic concerns. Finally, in Section 7 I relate my approach to structural equivalence and inclusion testing to other work in the field.

2 Name Equivalence versus Structural Equivalence

The difference between name equivalence and structural equivalence can be dramatized by the following code example from a strongly typed SCHEME-like [Rees & Clinger 86] language:

```
;; 2D-Matrix is a list of lists in row major form
(DEFINE-TYPE 2D-Int-Matrix  (LISTOF Int-Matrix-Row))
(DEFINE-TYPE Int-Matrix-Row (LISTOF INTEGER))

;; Payroll Database is a list of 2-tuples: salary X IDs
(DEFINE-TYPE Payroll-Database
             (LISTOF Payroll-Database-Entry))
(DEFINE-TYPE Payroll-Database-Entry
             (PAIROF Salary (LISTOF ID-Num)))
(DEFINE-TYPE Salary INTEGER)    ;; Employee salary
(DEFINE-TYPE ID-Num INTEGER)    ;; Employee ID Number

;; All-Paths is a list of paths where a path is an
;;  alternating list of a city and the distance to the
;;  next city on the path
(DEFINE-TYPE All-Paths (LISTOF Path))
(DEFINE-TYPE Path
             (RECTYPE p
               (PAIROF Site-Num (PAIROF Distance p))))
(DEFINE-TYPE Site-Num INTEGER)  ;; Numbered Sites
(DEFINE-TYPE Distance INTEGER)  ;; Distance to next site
```

In a name equivalence system, all three of the above types `2D-Int-Matrix`, `Payroll-Database`, and `All-Paths` are considered to be distinct types. Thus, for instance, were we to define a `matrix-transpose` procedure, we could not invoke it on a `Payroll-Database`. Likewise, were we to define a procedure to return a list of the `Salary` entries of a `Payroll-Database`, it could not be used to return a list of starting sites of an `All-Paths` list. With structural equivalence, by contrast, all of the above are possible. This is because we will adhere to the LISP tradition in our equivalence axioms in that a `LISTOF` type is equivalent to an arbitrary-length homogeneous chain of `PAIROF` cells.

Of course, advocates of name equivalence are quick to counter that name equivalence provides a means of type abstraction which makes it impossible to inadvertently pass a `2D-Int-Matrix` where a `Payroll-Database` is expected. As Cardelli has noted [Cardelli 88][2] [Cardelli *et al.* 89, §4], the proposed security afforded by name equivalence is better achieved by a proper abstract data type (ADT) mechanism which can properly hide the representation type and regulate access to it. Such an ADT facility is, for instance, provided in the languages CLU [Liskov *et al.* 77] [Liskov *et al.* 79/81], ADA [Barnes 80], and PEBBLE [Burstall & Lampson 84/88] [Lampson & Burstall 88]. A similar mechanism could be added to a structural equivalence system by extending it with new primitives to treat ADTs as `DEFINE-TYPE`s where the definition is never interrogated upon equivalence testing. In essence, ADTs should be compared via name equivalence but name equivalence *per se* does not provide ADTs.

[2] His arguments here are largely based on those in [Berry & Schwartz 79].

Hereafter, therefore, we will take "name equivalence" to refer to the discipline whereby only structurally distinct types will be given distinct names by the programmer, thus eliminating the issue of using name equivalence for weak ADTs. Thus, if a programmer using a structural equivalence language obeys the same discipline in naming of types as does a name equivalence programmer, then we should expect the time required to type check their programs to be nearly identical for a well-reasoned structural equivalence implementation. Of course, a programmer using a structural equivalence language can relax this discipline, taking advantage of the additional flexibility inherent in structural equivalence. In such cases, however, a well-reasoned approach should charge no more for this transgression than is justified by the cost of the particular structural equivalence axioms needed to demonstrate equivalence. I derive such a well-reasoned structural equivalence checking algorithm in the ensuing sections.

3 Type Equivalence and Inclusion Axioms

In this section, I introduce a kernel system of type expressions and the equivalence and inclusion axioms over these types. I believe this kernel to be sufficiently rich to dramatize the important issues while not being so largess as to obfuscate the essential points. For example, we hereafter abandon the `DEFINE-TYPE` form so that we needn't consider a global type environment, opting instead to consider only type expressions *en vacuo*. This allows us to avoid considering the syntax of value expressions in which type expressions would appear as type declarations. Extending our type system to include a type environment is straightforward. See for instance [Cardelli 89].

3.1 Descriptions: Types and DFUNCs

Beyond the type expressions alluded to already, we introduce `DFUNC` forms to make our language of type expressions itself comprise a full-fledged purely functional typed λ-calculus [Church 51] [Barendregt 84]. For example, `LISTOF` could be defined in terms of Dλ to be a `DFUNC` that maps types to types, as we shall see in Figure 1 [Section 3.3.3]. These generalized types are called *descriptions* (hence `DFUNC` abbreviates *description function*). Following McCracken [McCracken 84], a `KIND` is the "type" of a description. In general, our language would therefore include kind checking axioms on descriptions so that a `DFUNC` cannot appear in a well-formed description at a point where a `TYPE` is expected, but rather than enumerate the kind checking axioms here we rely instead on the suggestive meta-variable names in the grammar below to convey the kind constraints. Detailed kind checking axioms can be found, for example, in [Gifford *et al.* 87].

3.1.1 [Kinds]

κ ::= TYPE | (DFUNC (κ ...) κ)

3.1.2 [Descriptions]

δ ::= τ | δ_ψ | $\delta_?$

δ_ψ ::= (Dλ (($param$ κ) ...) δ) | $\delta_?$
$\delta_?$ ::= (δ_ψ δ ...) | $param$

τ ::= τ_{base} | τ_{aggr} | $\delta_?$

τ_{base} ::= BOOLE | INT | NULL | VOID
τ_{aggr} ::= (SUBR (τ ...) τ) | (PAIROF τ τ')
| (RECORDOF (tag τ) ...)
| (ONEOF (tag τ) ...)
| (RECTYPE $param$ τ)

$param$ ::= $identifier$
tag ::= $identifier$

This provides us with a type language as powerful as the pure typed λ-calculus with recursive data (the RECTYPE form). It was inspired by McCracken's language LS1 [McCracken 84], which was in turn based on Reynolds' second-order λ-calculus [Reynolds 74]. It also strongly resembles the type system of Cardelli's AMBER [Cardelli 84/85/86]. We cannot hope to do better than a strongly normalizing λ-calculus while maintaining a decidable type language.

3.2 Description Equivalence Axioms

3.2.1 [Basic Equivalence Axioms]

The following define $\equiv$ to be a congruence relation over description expressions.

$$\begin{array}{rcl} \delta & \equiv & \delta \\ \delta \equiv \delta' \;\wedge\; \delta' \equiv \delta'' & \Longrightarrow & \delta \equiv \delta'' \\ \delta \equiv \delta' & \Longrightarrow & \delta' \equiv \delta \\ \delta_\psi \equiv \delta'_\psi \;\wedge\; \forall i\,.\, \delta_i \equiv \delta'_i & \Longrightarrow & (\delta_\psi\ \delta_1 ... \delta_n) \equiv (\delta'_\psi\ \delta'_1 ... \delta'_n) \\ \delta \equiv \delta' & \Longrightarrow & \delta'' \equiv \delta''\{\delta'/\delta\} \end{array}$$

...where $\delta''\{\delta'/\delta\}$ denotes δ'' with the substitution of δ' for any or all occurrences of δ within δ''. Although this last rule (substitutivity) follows from the one just above it ("pointwise" $\equiv$), it is nonetheless included here for perspicuity. It will come in handy when discussing a equivalence decision procedure.

3.2.2 [Type Equivalence]

Pointwise equivalent descriptions are always equivalent. Pointwise inequivalent type expressions are never equivalent, the sole exception being that occurrences of (tag τ) binding pairs within ONEOFs and RECORDOFs is not order-specific. That is:

$$\begin{array}{rl} & (\texttt{ONEOF}\ (\texttt{tag}_1\ \tau_1)...(\texttt{tag}_n\ \tau_n)) \\ \equiv & (\texttt{ONEOF}\ (\texttt{tag}_{\pi(1)}\ \tau_{\pi(1)})...(\texttt{tag}_{\pi(n)}\ \tau_{\pi(n)})) \end{array}$$

$$\begin{array}{rl} & (\texttt{RECORDOF}\ (\texttt{tag}_1\ \tau_1)...(\texttt{tag}_1\ \tau_1)) \\ \equiv & (\texttt{RECORDOF}\ (\texttt{tag}_{\pi(1)}\ \tau_{\pi(1)})...(\texttt{tag}_{\pi(n)}\ \tau_{\pi(n)})) \end{array}$$

...for any permutation function π of the positive integers $1 \ldots n$. This is tantamount to saying that a RECORDOF/ONEOF form consists of a *set* of (tag τ) binding pairs, where the set of tags must be irredundant.

3.2.3 [Higher Order Description Equivalence]

The following constitute the familiar α, β and η rules of the λ-calculus, in that order:

$$\frac{\forall i\,.\ \texttt{dvar}'_i \notin \texttt{FDV}(\delta_{\text{body}}) \cup \{\texttt{dvar}_1 ...\}}{(\text{D}\lambda\ ((\texttt{dvar}_1\ \kappa_1)...)\ \delta_{\text{body}}) \equiv (\text{D}\lambda\ ((\texttt{dvar}'_1\ \kappa_1)...)\ \delta_{\text{body}}[\texttt{dvar}'_i/\texttt{dvar}_i])}$$

$$\begin{array}{rcl} ((\text{D}\lambda\ ((\texttt{dvar}_1\ \kappa_1)...)\ \delta_{\text{body}})\ \delta_1...) & \equiv & \delta_{\text{body}}[\delta_i/\texttt{dvar}_i] \\ & & \text{iff } \forall i\,.\ \delta_i \text{ has kind } \kappa_i \\ (\text{D}\lambda\ ((\texttt{dvar}_1\ \kappa_1)...)\ (\delta_\psi\ \texttt{dvar}_1\ ...)) & \equiv & \delta_\psi \\ & & \text{iff } \forall i\,.\ \texttt{dvar}_i \notin \texttt{FDV}(\delta_\psi) \end{array}$$

...where FDV is the Free Description Variables function, defined in the obvious way. (See, for instance, [Gifford *et al.* 87].)

3.2.4 [RECTYPE Equivalence]

A RECTYPE form denotes a recursive type.[3] Therefore a RECTYPE form is equivalent to any description which is pointwise equivalent to its (possibly) infinite unwinding, that is, if corresponding components of the two descriptions are equivalent. This is formalized by the following axiom:

$$(\texttt{RECTYPE t}\ \tau) \equiv \tau[(\texttt{RECTYPE t}\ \tau)/\texttt{t}]$$

3.2.5 [Completeness of the Equivalence Axioms]

Description expressions which cannot be proved equivalent by the above equivalence axioms are, by definition, not equivalent. Thus, descriptions which are distinct modulo the above equivalence axioms are not equivalent.

3.3 Description Inclusion Axioms

3.3.1 [Basic Inclusion Axioms]

The following define $\sqsubseteq$ to be an inclusion/partial-order relation over description expressions.

$$\begin{array}{rcl} & \delta \sqsubseteq \delta & \\ \delta \sqsubseteq \delta' \;\wedge\; \delta' \sqsubseteq \delta'' & \Longrightarrow & \delta \sqsubseteq \delta'' \\ \delta \sqsubseteq \delta' \;\wedge\; \delta' \sqsubseteq \delta & \Longleftrightarrow & \delta \equiv \delta' \end{array}$$

The final rule stipulates that inclusion respects equivalence. From these axioms one can deduce the following two substitutivity lemmas, which are useful in practice since they prove the soundness of a more "fully lazy" normalization strategy than the one proposed here:

$$\frac{\delta \equiv \delta'}{\delta \sqsubseteq \delta'' \implies \delta' \sqsubseteq \delta''} \qquad \frac{\delta \equiv \delta'}{\delta'' \sqsubseteq \delta \implies \delta'' \sqsubseteq \delta'}$$

For example, these two axioms imply that once two types are known to be equivalent, any memoized inclusion relations proven with respect to one of them and a third description can safely be inherited by the other.

[3] Note that we do not permit recursive DFUNCs, although we do permit DFUNCs whose bodies can be a RECTYPE.

3.3.2 [Inclusion Notational Convention]

For the following inclusion rules, I use a non-traditional notational convention that a term with a subscript '$\downarrow$' is constrained to be in $\sqsubseteq$ relation to the corresponding term with a subscript '$\uparrow$' on the other side of the inequality. For instance:

$$(\texttt{SUBR}\ (\tau_{1\uparrow}\ \ldots\ \tau_{n\uparrow})\ \tau_{\mathrm{rtn}\downarrow}) \sqsubseteq (\texttt{SUBR}\ (\tau_{1\downarrow}\ \ldots\ \tau_{n\downarrow})\ \tau_{\mathrm{rtn}\uparrow})$$

is shorthand for the more verbose

$$\frac{\begin{array}{rcl}\tau_{i\downarrow} & \sqsubseteq & \tau_{i\uparrow} \quad \forall i\ .\ 1 \le i \le n \\ \tau_{\mathrm{rtn}\downarrow} & \sqsubseteq & \tau_{\mathrm{rtn}\uparrow}\end{array}}{(\texttt{SUBR}\ (\tau_{1\uparrow}\ \ldots\ \tau_{n\uparrow})\ \tau_{\mathrm{rtn}\downarrow}) \sqsubseteq (\texttt{SUBR}\ (\tau_{1\downarrow}\ \ldots\ \tau_{n\downarrow})\ \tau_{\mathrm{rtn}\uparrow})}$$

Notice the anti-monotonicity in the argument lists. This makes the SUBR rule the most subtle of all the inclusion rules. Intuitively, if we view types as describing values and subtypes as subsets, and if we view a SUBR type τ as denoting a set of procedures which map from a domain to a range, then the set of procedures which are defined on a possibly larger domain but which always map to a possibly smaller range will be a subset of the denoted procedures. Thus, any SUBR type τ' which denotes only elements of this subset is a subtype of τ. The remaining inclusion rules are more straightforward.

3.3.3 [Type Inclusion]

$$\begin{array}{rcl}
\texttt{VOID} & \sqsubseteq & \tau \quad \forall\tau \\[1ex]
\texttt{NULL} & \sqsubseteq & (\texttt{PAIROF}\ \tau\ \ \tau') \quad \forall\tau,\tau' \\
(\texttt{PAIROF}\ \tau_{\downarrow}\ \tau'_{\downarrow}) & \sqsubseteq & (\texttt{PAIROF}\ \tau_{\uparrow}\ \tau'_{\uparrow}) \\[1ex]
(\texttt{SUBR}\ (\tau_{1\uparrow}\ \ldots\ \tau_{n\uparrow})\ \tau_{\mathrm{rtn}\downarrow}) & \sqsubseteq & (\texttt{SUBR}\ (\tau_{1\downarrow}\ \ldots\ \tau_{n\downarrow})\ \tau_{\mathrm{rtn}\uparrow}) \\[1ex]
(\texttt{ONEOF}\ \ (\texttt{tag}_1\ \tau_{1\downarrow})\ldots(\texttt{tag}_n\ \tau_{n\downarrow})) & \sqsubseteq & (\texttt{ONEOF}\ \ (\texttt{tag}_1\ \tau_{1\uparrow})\ldots(\texttt{tag}_n\ \tau_{n\uparrow})) \\
(\texttt{RECORDOF}\ (\texttt{tag}_1\ \tau_{1\downarrow})\ldots(\texttt{tag}_n\ \tau_{n\downarrow})) & \sqsubseteq & (\texttt{RECORDOF}\ (\texttt{tag}_1\ \tau_{1\uparrow})\ldots(\texttt{tag}_n\ \tau_{n\uparrow}))) \\[1ex]
(\texttt{ONEOF}\ \ (\texttt{tag}_1\ \tau_{1\downarrow})\ldots(\texttt{tag}_n\ \tau_{n\downarrow})) & \sqsubseteq & (\texttt{ONEOF}\ \ (\texttt{tag}_1\ \tau_{1\uparrow})\ldots(\texttt{tag}_n\ \tau_{n\uparrow}) \\
 & & \quad (\texttt{tag}'\ \tau'\ )\ldots(\texttt{tag}''\ \tau''\ )) \\[1ex]
(\texttt{RECORDOF}\ (\texttt{tag}_1\ \tau_{1\downarrow})\ldots(\texttt{tag}_n\ \tau_{n\downarrow}) & \sqsubseteq & (\texttt{RECORDOF}\ (\texttt{tag}_1\ \tau_{1\uparrow})\ldots(\texttt{tag}_n\ \tau_{n\uparrow})) \\
\quad (\texttt{tag}'\ \tau'\ )\ldots(\texttt{tag}''\ \tau''\ )) & &
\end{array}$$

Notice the peculiar rules for VOID and NULL. These state that VOID is the bottom of the TYPE partial order defined by $\sqsubseteq$ and that NULL is the bottom of the PAIROF partial order. There are no values of type VOID and there is a single value of type NULL, namely '(). This treatment of NULL enables us to define the traditional LISP notion of a list as a null-terminated pair. For instance, the type constructor LISTOF describing homogeneous[4] lists could be defined as in Figure 1 [Gifford *et al.* 87, p.66]:

```
LISTOF  =  (Dλ ((τ TYPE))
               (RECTYPE listof-τ (PAIROF τ listof-τ)))
```

Figure 1: Definition of LISTOF as a DFUNC

[4] A heterogeneous list can then be described as a homogeneous list comprised of ONEOF (tagged variant) elements. *E.g.,* (LISTOF (ONEOF (i INT)(b BOOLE))) describes a list of elements each of which is tagged with either i or b indicating whether it is of type INT or BOOLE, respectively.

For example, (PAIROF INT (PAIROF INT NULL)) $\sqsubseteq$ (LISTOF INT), so one can say that (CONS 4 (CONS 2 '())) is an instance of a (LISTOF INT).

The asymmetry in the latter RECORDOF and ONEOF rules deserves comment. RECORDOF forms describe vectors of data that are accessed by tag. Therefore, viewing types as sets and subtypes as subsets, the set of objects with tag fields in excess of a description is a subset of all objects having at least those tag fields. This is why the subtype RECORDOF is permitted to have more tag/type pairs than the supertype RECORDOF, so long as corresponding fields are in inclusion relation as usual. The ONEOF rule is just the reverse. A ONEOF form describes a single tagged value that may vary in value within the limits of the tag/type pairs stipulated. Therefore, again adopting the subtypes as subsets view, the set of values which are restricted from varying outside a set of tag/type pairs are a subset of those values which are restricted from varying outside a larger set of tag/type pairs, with the usual correspondence inclusion stipulation.

3.3.4 [Higher Order Description Inclusion]

The first rule below permits Dλ inclusion in the obvious way. The second rule, by contrast, stipulates that inclusion of non-β-reducible description applications is by equivalence: because kind information carries no information about whether a DFUNC is monotonic or anti-monotonic in any particular description argument, it is not safe to allow the description application operator δ_ψ to be in inclusion relation. Moreover, since the operator "tonicity" information is unavailable, inclusion on the operands would likewise be unsafe. This last rule could be deduced from the axioms which constrain inclusion to respect equivalence; it is included here only to draw attention to this tonicity issue. Finally, note [§ 3.1.2] that any description application operator δ_ψ in normal form will be either an identifier or will itself be of the form $(\delta_\psi\ \delta\ \ldots)$.

$$\begin{array}{rcl}
(\mathrm{D}\lambda\ ((\texttt{dvar}\ \kappa)\ldots)\ \delta_{\mathrm{body}\downarrow}) & \sqsubseteq & (\mathrm{D}\lambda\ ((\texttt{dvar}\ \kappa)\ldots)\ \delta_{\mathrm{body}\uparrow}) \\
(\delta_\psi\ \delta\ \ldots) & \sqsubseteq & (\delta_\psi\ \delta\ \ldots)
\end{array}$$

3.3.5 [RECTYPE Inclusion]

As with equivalence, a RECTYPE form is included in another description iff its (possibly) infinitely unwound body is "pointwise" included in the description, that is, if corresponding components are in the appropriate (monotonic or anti-monotonic) inclusion relation. This is formalized by the following sequent rules, where $\mathcal{R}$ denotes a set of inclusion constraints on RECTYPE parameters and where $\mathcal{S} \vdash_{\sqsubseteq} \mathcal{I}$ means "Given a set $\mathcal{S}$ of inclusion assumptions, the inclusion relation $\mathcal{I}$ can be proven using the inclusion axioms."

$$\frac{\mathcal{R} \cup \{\texttt{t} \sqsubseteq \tau_\uparrow\} \ \vdash_{\sqsubseteq}\ \tau_\downarrow \sqsubseteq \tau_\uparrow}{\mathcal{R}\ \vdash_{\sqsubseteq}\ (\texttt{RECTYPE t}\ \tau_\downarrow) \sqsubseteq \tau_\uparrow}$$

$$\frac{\mathcal{R} \cup \{\tau_\downarrow \sqsubseteq \texttt{t}\} \ \vdash_{\sqsubseteq}\ \tau_\downarrow \sqsubseteq \tau_\uparrow}{\mathcal{R}\ \vdash_{\sqsubseteq}\ \tau_\downarrow \sqsubseteq (\texttt{RECTYPE t}\ \tau_\uparrow)}$$

4 The Induced Search-Space Lattices

4.1 Equivalence as a Reduction System

These equivalence and inclusion axioms over descriptions define a term rewrite system semantics of descriptions. Focusing for the moment on the equivalence axioms, one can derive a reduction semantics for descriptions by noting that these axioms can each be viewed as reduction rules. Specifically, the RECORDOF and ONEOF axioms can be viewed as sorting axioms based on field tags. The tacit pointwise equivalence axioms on SUBR and PAIROF types can be viewed as saying that they are reduced componentwise.

Of the familiar λ-calculus axioms, the β and η rules have their standard left-to-right reduction orientation. The α-renaming axiom can be made into a reduction axiom by noting that description parameters can be renamed in a systematic way, for instance, by using a variant of DeBruijn numbering [de Bruijn 72] or lexical addressing [Abelson & Sussmans 85] [Ward & Halstead 89].

The RECTYPE equivalence axiom is more subtle. The insight is that RECTYPE expressions can be modeled as expressions from the μ-calculus [Park 76], from which μ-calculus minimization can be used as a reduction strategy. In effect, the process involves re-encoding RECTYPE expressions into directed acyclic graphs (dags)[5] from which DFA-minimization can be performed to minimize the number of RECTYPE parameters introduced by collapsing equivalent subexpressions. Afterward, normalization of the parameter names can be performed, for instance, by numbering parameters according to their first occurrence in a left-to-right traversal of the minimized RECTYPE expression in topological order.

The advantage of viewing the equivalence axioms as inducing a reduction semantics on descriptions is the observation that this reduction system reduces descriptions to a unique equivalence class representative (a *normal form*). Consequently, two types will be equivalent iff they reduce to textually identical normal forms. In effect, *structural equivalence can be reduced to a mere reflexivity test on normalized descriptions.* This, for instance, is precisely the approach taken in [Jouvelot & Gifford 88]. The problem with such an approach, of course, is that all applicable axioms are actually applied in reducing a type to normal form rather than just that subset of the axioms actually necessary to demonstrate the equivalence of two specific descriptions. For example, in [Jouvelot & Gifford 88], two Dλ forms whose bodies are recursive RECORDOFs whose bodies textually differ only in their Dλ-bound parameter name would be fully normalized when compared for equivalence. This entails both RECORDOF field sorting and RECTYPE minimization. Indeed, in an early version of the system, even textually identical descriptions would be fully normalized.

The important observation here is that *there may be no need to do full normalization in order to demonstrate equivalence of two descriptions.* Rather, we need only partially normalize the compared descriptions to the extent that they become textually identical, albeit not necessarily to their full normal form. Moreover, since the reduction to a normal form is a confluent operation (it has the "diamond" property of Church-Rosser [Rosser 82]), it is irrelevant in what order we choose to apply reduction axioms to the descriptions being compared.

On the other hand, if two types are not equivalent we will not notice this until we have fully normalized both types. This is not problematic, however, since equivalence will only fail in one of two situations: 0) the program is in error, so we don't care how much effort we expend in eventually discovering this (we want to be optimistic), or 1) strict inclusion is being employed by the programmer. As we shall see, when we must resort to testing inclusion, having already fully normalized the two types is a valuable asset.[6]

4.2 The Induced Normalization Lattice

Viewing the equivalence axioms as constituting a reduction semantics, graphically, as exemplified by Figure 2, these reduction rules give rise to a (downward directed) normalization lattice, where the source type expression is at the top, each branch corresponds to a single application of one reduction rule, and the bottom is the normal form of the source type expression.

```
                (ONEOF (A ((t → t) a)) (B ((t → t) b)))
                              //        \\
                             //          \\
(ONEOF (A a) (B ((t → t) b)))  (ONEOF (A ((t → t) a)) (B b))
                             \\          //
                              \\        //
                          (ONEOF (A a) (B b))
```

Figure 2: A Simple Normalization Lattice

Now we can view the reduction to normal form as a downward traversal along a normalization lattice. When we compare two type expressions for equivalence, we then need only traverse their respective normalization lattices to their earliest point of intersection.[7] For instance, in the example above if we were comparing the original source type expression with, say, the left type expression on the second reduction level, then we could avoid reducing them both to the bottom normal form. When two type expressions are in fact not equivalent, then that too will be discovered since their normalization lattices will never intersect.

Note that the above is not meant to suggest that the full normalization lattice actually ever be generated; rather, a comparison procedure need only simulate traversal over this lattice. That is, comparison can be viewed as a search problem over a pair of normalization lattices for the first point of intersection of these downward directed lattices, where the search lattices can in effect be generated on-the-fly.

This is an entirely natural way to verify equivalence. Algebraically, all that is necessary to prove equivalence is that the two sides of an equation be made textually identical. This approach works so long as we are restricted to axioms that do not allow expressions to be transferred from one side of the equation to the other.[8]

[5] This entails the μ-calculus analog of super-combinator λ-lifting [Peyton-Jones 87].

[6] More than just reflecting a happy coincidence, I suspect this actually reflects some deep property of equivalence/inclusion systems for which inclusion respects equivalence in the sense of the base inclusion axioms [Section 3.3.1].

[7] The earliest point of intersection is just the greatest lower bound (glb) of the compared descriptions with respect to the universal description normalization lattice.

[8] An example of an axiom that violates this restriction is the α-renaming rule, which states that two function types which differ only in their choice of parameter names can be made equivalent by renaming the parameters of the one type to match the corresponding names of the other. To make our normalization of a type be independent of any type it is being compared against, we choose instead to systematically rename the parameters of both types.

4.3 The Induced Inclusion Lattice

Switching our attention now from equivalence to the more general problem of inclusion, we could imagine a new lattice denoting the lattice of subtypes of a given type expression. For example, the inclusion lattice of the type expression (ONEOF (A a) (B b) (C c)) is depicted in Figure 3.

```
                    (ONEOF (A a) (B b) (C c))
                           /    |    \
                         /      |      \
                       /        |        \
                     /          |          \
                   /            |            \
(ONEOF (A a) (B b))    (ONEOF (A a)(C c))    (ONEOF (B b) (C c))
              |  \          /    \          /  |
              |    \      /        \      /    |
              |      \ /             \ /       |
              |      / \             / \       |
              |    /     \         /     \     |
              |  /         \     /         \   |
      (ONEOF (A a))       (ONEOF (B b))       (ONEOF (C c))
                   \            |            /
                     \          |          /
                       \        |        /
                         \      |      /
                           \    |    /
                            (ONEOF)
```

Figure 3: A Simple Inclusion Lattice

For most type expressions, this is just the powertype [Cardelli 88] (powerset) lattice with the source type expression as the top. That is, if we consider types to denote sets of possible values, then subtypes denote subsets of these possible values; hence the set of all subtypes of a type expression denotes the powerset of possible values, assuming the type language to be rich enough to express all such subsets [Cardelli 87].

For other type expressions (like SUBR types) which do not have an obvious subset-like algebraic textual appearance, the visual analogy to a powerset is less immediate for the type expressions themselves, yet they represent precisely this sort of powerset over possible values.

Conceptually then, verifying inclusion is tantamount to traversing the inclusion lattice of the presumed supertype, searching for a node equivalent to the presumed subtype. As with the normalization lattice, this inclusion lattice too could be generated on-the-fly by the inclusion testing procedure.

5 Equivalence/Inclusion as Search

Characterizing equivalence and inclusion as inducing normalization and inclusion lattices, respectively, enables us now to consider competing strategies for equivalence/inclusion decision procedures as competing search strategies over this induced lattice search space.

5.1 Distance Metrics to Guide Comparison

First, when checking equivalence of two types, selecting the branch of the normalization lattice which is most efficient to pursue relies on having some reliable distance measure to the types' point of intersection (the glb). This measure, in turn, depends upon the specific two type expressions being compared, for example knowing that tag field sorting is unnecessary for two RECORDOF forms with the same tag field order. Thus, an accurate measure requires at least partial comparison of the two expressions involved.

For instance, a greedy algorithm might perform a breadth-first search over two type expressions to find the first pair of corresponding subexpressions which are not textually identical. For these two subexpresions, the equivalence axiom system could be polled for those axioms which would reduce their distance, for instance by reducing one of them to a reduct more structurally similar to the other or, in lieu of that, by reducing both of them to more structurally similar type expressions. The comparison traversal could then proceed by choosing one of the paths corresponding to the application of one of these favorable axioms.

This clearly requires a great deal of specific knowledge about the axiom system and might indeed involve more computational overhead in computing distances than a straightforward reduction to normal form might take. Nonetheless, this notion of a heuristic distance to a glb will surface again later.

5.2 The Induced Normalization/Inclusion Cubic Lattice

Second, it has not yet been made clear how the normalization lattice interacts with the inclusion lattice. Specifically, casual reference was made to performing inclusion testing by traversing the inclusion lattice of the presumed supertype in search of a node *equivalent* to the presumed subtype. This suggests some intermingling of the normalization and inclusion lattices. Specifically, each node of the normalization lattice is associated with an entire inclusion lattice for that particular degree of normalization of the expression. For example, reconsider the ONEOF type normalization lattice of Figure 2:

```
          (ONEOF (A ((t → t) a)) (B ((t → t) b)))
                        //        \\
                       //          \\
(ONEOF (A a) (B ((t → t) b)))  (ONEOF (A ((t → t) a)) (B b))
                       \\          //
                        \\        //
                   (ONEOF (A a) (B b))
```

Each node of this normalization lattice corresponds to the root of an inclusion lattice consisting of the ONEOF powerset over the fields A and B at that node. For example, the bottommost term of this normalization lattice corresponds to the leftmost term on the second level of the inclusion lattice in Figure 3. Thus, it is the root of the inclusion sublattice starting at that node.

This intermingling of normalization and inclusion lattices can be viewed, in general, as a three-dimensional lattice with normalization and inclusion forming independent axes and choice of axiom[9] forming the third.

Graphically, we could envision normalization lattices as positioned in the horizontal plane, with their normal form positioned leftmost. Inclusion lattices would then be vertically positioned with the least element at the bottom. The sketch in Figure 4 below is intended to be roughly suggestive of this arrangement, where ∪ is used to abbreviate ONEOF.

[9] That's *choice of axiom*, not Axiom of Choice.

```
        -------------------------------------------------------------------
       /                                                                 /
      / (∪ (A ((t → t) a)) (B b))⇐==(∪ (A ((t → t) a)) (B ((t → t) b))) /
     /          ||        ^               ||          ^                /
    /           ⇓        / \              ⇓          / \              /
   /     (∪ (A a) (B b))⇐===(∪ (A a) (B ((t → t) b)))/   \           /
  /               ^    /     \   /               \   /    \         /
  -------------------------------------------------------------------
      ----------/---/---------\--------------------\---------\------------
     /         /   / \      /  \                  / \         \         /
    /         /   / (∪ (A a))⇐(∪ (A ((t → t) a)))  \          \       /
   /         /   /        :                :         \          \     /   [THIS WILL BE DRAWN WITH A REAL GRAPHICS EDITOR...
  /       (∪ (B b))⇐==================(∪ (B ((t → t) b)))  /
 /            :           :           :            :        /     ...IN THE FINAL DRAFT]
 -------------------------------------------------------------------
     -------------:-------:-------:---------:------------------------
    /                 :      :      :       :                      /
   /                    :   :   :   :                             /
  /                          (∪)                                 /
 /                                                              /
 ----------------------------------------------------------------
```

Figure 4: A Sample Normalization/Inclusion Cubic Lattice

5.3 Inclusion Testing as Search

Now, were we to naïvely navigate this cubic lattice upon testing inclusion of two type expressions, inclusion could become quite expensive indeed. For example, were we to traverse the inclusion lattice of the unnormalized presumed supertype expression, at each step testing afresh if this new unnormalized type is equivalent to the unnormalized presumed subtype, then this could become exponentially expensive.

Graphically, this would correspond to descending the inclusion lattice of the source supertype expression (the rightmost vertical face of Figure 4), at each node exhausting the corresponding equivalence lattice at that level (the horizontal face at that level) to determine if we are yet equivalent to the presumed subtype, which also entails matching against the entire presumed subtype's normalization lattice.

5.4 A Proposed Inclusion Traversal Strategy

Alternatively, we could test inclusion by first testing the type expressions for mere equivalence. In the case where they cannot be proven equivalent, we will proceed to reduce both of them to normal form (that is, traverse both normalization lattices to their leftmost extremities). We would then proceed with testing for inclusion by descending the inclusion lattice of the normalized presumed supertype, testing at each step for equivalence with the normalized presumed subtype.

Graphically, this corresponds to lockstep traversal of the normalization lattices of the two types being compared (the topmost faces of the two normalization/inclusion cubic lattices). Upon reaching the leftmost nodes of each normalization lattice without discovering equivalence, the traversal proceeds by descending the inclusion lattice of the now-normalized presumed supertype, testing equivalence of each node against its corresponding node in the now-normalized presumed subtype.

Using this strategy, inclusion testing could in the worst case traverse only two faces of the supertype's normalization/inclusion cubic lattice— namely, the topmost horizontal (normalization) face and the leftmost vertical (inclusion) face— and one face of the subtype normalization/inclusion lattice— namely, the topmost horizontal (normalization) face.

Many alternative traversal strategies are possible, but this one seems particularly well motivated in view of the possibility of memoizing the normalization of a description, for instance by employing hash-`CONS`ing [Field & Harrison 88] in type expressions. Also, if a programmer makes little use of the availability of general structural inclusion, then we can expect mere equivalence testing to usually suffice where inclusion is permitted. In fact, if a programmer adheres to name equivalence, then the only axiom which ever need be consulted is the trivial reflexivity axiom.

Returning to our proposed inclusion traversal strategy, even with the above pre-normalization of inclusion candidates, it is still unclear how to direct the inclusion lattice search when the easier equivalence test has failed. Here, however, a distance metric on which inclusion path to pursue is more straightforward given that the two expressions will be in normal form. Specifically, the equivalence axioms need no longer be considered, so inclusion testing can be implemented as a structural lockstep traversal of the normalized presumed supertype and subtype, recursively checking for pointwise inclusion of the respective components according to the applicable inclusion axioms. For example, inclusion traversal of two `PAIROF` types would recursively test the inclusion of the left and right pair subexpressions. With each recursive pointwise inclusion test, invoking unnecessary inclusion axioms could be avoided by testing for equivalence first. This is particularly frugal since, given that all subexpressions of a normal form will themselves be in normal form, no additional equivalence axioms need be invoked: *a trivial reflexivity test suffices.*

Another concern in devising an inclusion traversal strategy is that some type expressions (notably `RECORDOF` types) may have a countably infinite number of subtypes. For example, where a record with two fields is expected, a record with twenty fields could be passed so long as two of the twenty fields denote compatible components (relative ordering of fields is irrelevant). A record with two hundred fields could similarly be passed for the twenty-field record, and so on to arbitrary depth. Alternatively, a record with only one additional field is also compatible with a record without the extra field, where the type associated with this extra field can be any of a countably infinite set of choices. Thus, the `RECORDOF` inclusion lattice has infinite fan-out as well as infinite depth. Obliviously traversing such an inclusion lattice, exhaustively searching for the occurrence of some expression which may never appear (*e.g.,* when the presumed subtype in fact is not a subtype of the presumed supertype) would result in a potentially divergent type checker. Fortunately, the preceding proposal to test inclusion axioms via structurally recursive lockstep traversal of the two types being compared accommodates comparison against these potentially troublesome infinite-subtype descriptions.

Specifically, rather than incorporating the inclusion axioms into a database of axioms from which we poll for an inclusion axiom at each step (as we did with the equivalence axioms), we instead implement the inclusion axioms as Boolean predicates (inclusion testing subroutines) called from the inclusion testing procedure. That is, rather than use type rewrite rules which blithely wander the inclusion lattice in search of a reduct equivalent to the presumed subtype, we, for example, embody the `RECORDOF` inclusion axioms into a procedure which directs the search based on the two `RECORDOF`s being compared. This procedure, in essence, iterates over the fields of the presumed supertype testing for a compatible field in the presumed subtype.

As a final matter, the traversal strategies presented so far reek of redundant computations. Some memoization or hashing techniques would certainly be justified. For instance, computing the normal form of a specific occurrence of some type expression should be done at most once. Similarly, equivalence/inclusion findings should be memoized.

5.5 Equivalence Testing as Search

The pivotal unaddressed difficulty remains: How to supply a distance measure to guide the equivalence search through the normalization lattice when seeking the glb of two type expressions. Above, we argued that using the two expressions themselves to direct the search is unsavory. Instead, however, we can still use the axiom system itself to direct the normalization. Specifically, we can associate with each equivalence axiom the computational complexity of applying that axiom. Thus, the distance between a description and its one-step reduction via some axiom can be the computational complexity of applying that axiom to the source description to produce the reduct. This distance metric based on algorithmic complexity of the axioms can serve to direct our normalization efforts, as described below.

5.6 A Proposed Equivalence Traversal Strategy

We now adopt the view of equivalence testing as traversing the normalization lattices of the two types involved. Given thatour equivalence testing procedure will generate these normalization lattices on-the-fly, it is paramount that we chose an inexpensive traversal path, making inexpensive reflexivity tests along the way.

This incremental traversal/generation of the normalization lattice is tantamount to incrementally reducing a description to nomal form. If we now concentrate on incrementally reducing a type expression to its normal form by making several passes over the expression, each pass more computationally complex than the last, then after each complete pass we will have a type expression closer to normalization and we will have generated it within a given complexity bound. For instance, we could endeavor to exhaust all the linear-time axioms on the first pass, the $O(n \lg n)$ axioms on the second, the $O(n^2)$ on the third, and so on.

Specifically, of the axioms presented here: the β and η reductions are $O(n)$ (as is α-normalization by deBruijn labeling); sorting `RECORDOF` and `ONEOF` fields involves a comparison-based sort, which is $O(n \lg n)$; and `RECTYPE` normalization entails re-encoding into a DFA (which can be done in linear time) followed by DFA minimization, which is $O(\alpha\, n \lg n)$ where α is the size of the input alphabet [Aho & Ullman 86, §3.9, pp.141–144] [Hopcroft 71] [Aho, Hopcroft & Ullman 74, §4.13 Partitioning, pp.157–162]. For our encoding, α is the arity of the highest arity subexpression, which in practice can be assumed to be a constant probably less than about 8.[10]

Of course, in general, some axioms may need to be reapplied in subsequent phases if some later (more expensive) axiom gives rise to new reducts of an earlier (less expensive) axiom, but this will be absorbed into the cost of the more later expensive axioms.[11]

5.7 Formal Algorithms and Running Time Analysis

Formal details of equivalence and inclusion testing for the language FX-87 may be found in [Blair 90]. Specifically, Chapter 3 of that work presents a detailed gradual normalization strategy. Chapter 4 details the equivalence and inclusion algorithms with total correctness arguments and running time analysis.

6 Summary and Closing Remarks

6.1 Algorithmic Complexity Slicing

In our lazy, incremental normalization strategy, two type expressions are compared for equivalence by first testing if they already satisfy the trivial reflexivity axiom (pointer equality). If they do not, we apply the first complexity class of axioms (linear time) to each one, afterward checking for convergence to their glb by again testing reflexivity. So long as they fail to converge, we continue with the next complexity class of axioms. Note that since each subsequent phase will potentially dominate in running time over the previous phases, any unnecessary application of less expensive axioms in the earlier phases can be absorbed into the expense of the necessary subsequent more expensive axioms. Further, note that the most expensive situation is when type expressions involving the most expensive axioms are found to be, in fact, inequivalent; yet then they will have been fully reduced to normal form, the cost of which can be amortized over the

[10] Details of a design for using this approach in the FX-87 programming language is explored in a forthcoming technical report based on [Blair 90].

[11] Not surprisingly, the strategy used in [Blair 90] preserves the normalization of prior complexity-class equivalence axioms. This I suspect is a consequence of confluence in the reduction lattice. In addition, however, internal decorations within partially normalized descriptions are used there to accelerate both subsequent normalization phases and potential subsequent inclusion tests. Unfortunately, these internal state annotations are not in general preserved across partial reduction boundaries.

subsequent inclusion comparison, recalling that when equivalence search has failed we proceed with inclusion search.

In effect, we demarcate the normalization lattice with contour lines representing the algorithmic complexity of reducing the source description to the partially normalized description at the contour line. Our gradual normalization to normal form corresponds to a greedy algorithm which tries to cross these contour lines least expensive first. This is used by the equivalence testing procedure to gradually reduce the two compared descriptions to normal form only to the degree that they become textually identical. In the case where they fail to be equivalent, a more general structural inclusion test is performed. By that time, the descriptions being compared will have been fully normalized so inclusion proceeds looking only at the two fully normalized descriptions.

This technique of gradual normalization via increasingly complex partial normalization phases separated by early termination tests I call *algorithmic complexity slicing.* This approach is vaguely reminiscent of gradual deepening techniques common in game tree searches in the way in which conceptual complexity contours are drawn and in the way the cost of shallow searches are effectively absorbed into the cost of deeper ones.

6.2 Acceleration through Early Termination

Naturally, a variety of algorithmic acceleration techniques can be applied to this technique. For instance, type expressions can be represented via tupled data structures containing the memoized results of successive normalization slice boundaries. Additionally, these memoized intermediates can be hashed/interned to accelerate subsequent reflexivity tests. Moreover, the intermediates can themselves benefit from data structuring and annotation/decoration to facilitate subsequent inclusion axioms and inclusion tests. For example, internally decorating descriptions during the linear phaseof normalization can direct subsequent phases and can facilitate early termination of inapplicable phases. Specifically, annotating the free description variables of a description can accelerate β-reduction by avoiding a traversal of closed subexpressions. It also supports early termination since two descriptions that are known to contain different sets of free variables cannot be equivalent until they have been reduced into having equivalent free variables.

Also, the degree to which a type expression has been partially normalized can be annotated along with the intermediates so that whenever two types are proved equivalent, the less normalized type can adopt the intermediates of the more normalized type. This normalization inheritance is a variation on Tarjan's familiar Union/Find algorithm for determining equivalence classes [Tarjan 83]. An analogous adaptation applies to gradually generating the inclusion lattice relating all top-level defined types.

6.3 Closing Remarks

This approach has been incorporated into a design for the front-end of a compiler for the language FX-87 [Lucassen 87] [Gifford *et al.* 87] [Lucassen & Gifford 88]. The algorithms for actually performing equivalence and inclusion testing are detailed in [Blair 90], as well as consideration of how these algorithms can be incorporated into a type checking procedure. Unfortunately, the implementation of this design is not complete and too few programs have been written in FX-87 to provide empirical evidence of how beneficial this approach is in practice. It is my belief that many types introduced by a programmer will not be used in a subtyping setting which would require full normalization. I also believe, however, that in an object-oriented programming system setting, large recursive `RECORDOF` types that widely employ inclusion will be pervasive. In such settings, being able to dispense with the equivalence axioms when testing strict inclusion by having lazily memoized the applicable equivalence axioms into the normalized type expressions should provide a noticeable improvement on the running time of the inclusion testing algorithm.

7 Related Work

Other work in structural equivalence systems has been sparse, ALG L 68 [van Wijngaarden *et al.* 76] was the first to consciously choose to provide structural equivalence rather than name equivalence. Unfortunately, the formulation of recursive types in ALG L 68 allows recursive type *functions*, so the type language is not strongly normalizing [Coucelle 83]. Therefore, full normalization as a memoized consequence of performing equivalence testing is not realizable in ALG L 68 for types which contain `RECTYPE` components. Nonetheless, the algorithm used for equivalence testing of recursive types in ALG L 68 [van Wijngaarden *et al.* 76, §7.3, pp.103–107] provides the basis of the recursive type inclusion algorithm used in [Blair 90]. A similar approach to that taken in ALG L 68 has independently been taken in AMBER [Cardelli 90em].

Finally, as mentioned earlier, a normalization-based approach for equivalence testing was taken by Jouvelot in his experimental implementation of FX-87 [Jouvelot & Gifford 88]. His approach was to view the equivalence and inclusion axioms as comprising an abstract interpretation [Nielson 86] [Nielson 87/88] of descriptions over which one determines compatibility. Although memoization techniques are employed to avoid recomputation, no effort is made to avoid applying equivalence and inclusion axioms which are not necessary to demonstrate the particular compatibility test at hand. Consequently, descriptions are *always* reduced to full normal form there. This merely fuels the myth that structural equivalence is inherently more expensive than name equivalence. I hope that the strategy presented here will serve to dispell this myth.

Acknowledgments

I am indebted to Franklyn Turbak for helping me to organize these ideas. I have also benefited from many useful discussions with FX-perts Pierre Jouvelot, James O'Toole, and Mark Sheldon. Most importantly, my thesis advisor Prof. David A. McAllester was instrumental in helping me solidify these ideas. Specifically, it was he who called to my attention the correspondence between `RECTYPE`s and dags (DFAs). Early versions of this paper were reviewed by Bill Rozas, Arthur Gleckler, and Andy Ayers, each of whom provided many constructive and helpful comments. Despite the valiant efforts of all these people, any errors still contained herein are solely my own responsibility.

References

[Abelson & Sussmans 85]
Harold Abelson and Gerald Jay Sussman with Julie Sussman
Structure and Interpretation of Computer Programs (Fifth Printing)
(The MIT Electrical Engineering and Computer Science Series)
The MIT Press, 1985

[Aho, Hopcroft & Ullman 74]
Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman
The Design and Analysis of Computer Algorithms
Addison-Wesley, 1974

[Aho & Ullman 86]
Alfred V. Aho and Jeffrey D. Ullman
Compilers: Principles, Techniques and Tools
Addison-Wesley, 1986

[Barendregt 84]
H. P. Barendregt
The Lambda Calculus: Its Syntax and Semantics [Revised Edition]
Studies in Logic and Foundations of Mathematics Vol.103,
Elsevier Science Publishing Company, 1984

[Barnes 80]
J. G. P. Barnes
An Overview of ADA
Software — Practice and Experience, Vol.10, Jul 1980, pp.851–887

[Berry & Schwartz 79]
Daniel M. Berry and Richard L. Schwartz
Type Equivalence in Strongly Typed Languages: One More Look
In ACM SIGPLAN Notices, Vol.14, No.9, Sep 1979

[Blair 90]
Michael R. Blair
FIX: A Design for an Efficient Type/Region/Effect/Kind Checking System
MIT S.M. Thesis. To Appear as an MIT LCS/AI Technical Report.

[de Bruijn 72]
N. G. de Bruijn
Lambda Calculus Notation with Nameless Dummies, A Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem
In Proc. Akademie van Wetenschappen, Series A: Mathematical Sciences, Vol.75, pp.381–392, or
Indag. Math., Vol.34, 1972, pp.381–392

[Burstall & Lampson 84/88]
R. Burstall and B. Lampson
A Kernel Language for Modules and Abstract Data Types
DEC SRC TR No.1, Sept 1984

[Cardelli 84/85/86]
Luca Cardelli
AMBER
Tech. Memo 11271-840924-10TM, AT&T Bell Laboratories, Murray Hill, NJ, Sept 1984 or
In Proc. Treizième Ecole de Printemps d'Informatique Théorique, Le Val D'Ajol, Vosges (France), May 1985, pp.1–27 or
Combinatorics and Functional Programming Languages, Lecture Notes in Computer Science No.242, Springer-Verlag, 1986, pp.21–40

[Cardelli 84/88]
Luca Cardelli
A Semantics of Multiple Inheritance
In Semantics of Data Types, Lecture Notes in Computer Science No.173, Springer-Verlag, Jun 1984, pp.301–315 or
Information and Computation, Vol.76 Nos.2/3, Feb/Mar 1988, pp.138–164 (Updated)

[Cardelli & Wegner 85]
Luca Cardelli and Peter Wegner
On Understanding Types, Data Abstraction, and Polymorphism
In CACM Computing Surveys, Vol.17 No.4, Dec 1985, pp.471–522

[Cardelli 87]
Luca Cardelli
Typechecking Dependent Types and Subtypes
In Proc. Workshop on Foundations of Logic and Functional Programming, Trento, Italy, Dec 1986, Lecture Notes in Computer Science No.306, Springer-Verlag, 1986, pp.45–57

[Cardelli 88]
Luca Cardelli
Structural Subtyping and the Notion of Power Type (Extended Abstract)
In Proc. ACM POPL 1988, pp.70–79

[Cardelli 89]
Luca Cardelli
Typeful Programming
DEC SRC TR No.45, May 1989

[Cardelli *et al.* 89]
Luca Cardelli, Jim Donahue, Mick Jordan, Bill Kalsow and Greg Nelson
The M DULA 3 *Type System*
In Proc. ACM POPL 1989, pp.202–212

[Cardelli 90em]
Luca Cardelli
Private electronic mail correspondence 5 Mar 1990

[Church 51]
Alonzo Church
The Calculi of Lambda-Conversion
Annals of Mathematical Studies Vol.6, Princeton University Press, 1951

[Cook, Hill & Canning 90]
William R. Cook, Walter L. Hill, and Peter S. Canning
Inheritance is Not Subtyping
In Proc. ACM POPL 1990, pp.125–135

[Coucelle 83]
B. Coucelle
Fundamental Properties of Infinite Trees
In Theoretical Computer Science, Vol.25 No.2, Mar 1983, pp.95-169

[Field & Harrison 88]
Anthony J. Field and Peter G. Harrison
Functional Programming
(International Computer Science Series)
Addison-Wesley, 1988

[Gifford & Lucassen 86]
David K. Gifford and John M. Lucassen
Integrating Functional and Imperative Programming
In Proc. 1986 ACM Conference on LISP and Functional Programming, Aug 1986, pp.28–38

[Gifford *et al.* 87]
David K. Gifford, Pierre Jouvelot, John M. Lucassen, and Mark A. Sheldon
FX-87 *Reference Manual*
MIT/LCS/TR-407, Sept 1987

[Hopcroft 71]
John E. Hopcroft
"An nlgn algorithm for minimizing states in a finite automaton"
in *Theory of Machines and Computations* P. Kohavi and A. Paz (editors)
Academic Press, 1971, pp.189–196

[Jouvelot & Gifford 88]
Pierre Jouvelot and David K. Gifford
The FX-87 *Interpreter*
In Proc. IEEE Int. Conf. on Comp. Lang., Oct 1988, pp.65–72

[Lampson & Burstall 88]
B. Lampson and R. Burstall
PEBBLE, *A Kernel Language for Modules and Abstract Data Types*
Information and Computation, Vol.76 No.2/3, Feb/Mar 1988, pp.278–346

[Liskov *et al.* 77]
Barbara Liskov, Alan Snyder, Russel Atkinson, and Craig Schaffert
Abstraction Mechanisms in CLU
In CACM, Vol.20 No.8, Aug 1977, pp.564–575

[Liskov *et al.* 79/81]
Barbara Liskov *et al.*
CLU *Reference Manual*
MIT/LCS/TR-225, 1979 or
Lecture Notes in Computer Science No.114, Springer-Verlag, 1981

[Lucassen 87]
John M. Lucassen
Types and Effects: Towards the Integration of Functional and Imperative Programming
Ph.D. Thesis, MIT/LCS/TR-408, Aug 1987

[Lucassen & Gifford 88]
John M. Lucassen and David K. Gifford
Polymorphic Effect Systems
In Proc. ACM POPL 1988, pp.47–57

[McCracken 84]
Nancy McCracken
The Typechecking of Programs with Implicit Type Structure
In Semantics of Data Types. Lecture Notes in Computer Science No.173,
Springer-Verlag, Jun 1984, pp.301–315

[MacLennan 87]
Bruce J. MacLennan
Principles of Programming Languages: Design, Evaluation, and Implementation [2nd Edition]
CBS College Publishing, 1987

[Nielson 86]
Flemming Nielson
Abstract Interpretation of Denotational Definitions (A survey)
In STACS '86, Lecture Notes in Computer Science 210, Springer-Verlag, 1986, pp.1–20

[Nielson 87/88]
Flemming Nielson
Strictness Analysis and Denotational Abstract Interpretation (Extended Abstract)
In Proc. ACM POPL 1987, Jan 1987, pp.120–131, as condensed from
Information and Computation, Vol.76 No.1, Jan 1988, pp.29–92.

[Park 76]
David Park
Finiteness is μ-Ineffable
Theoretical Computer Science No.3, pp.173–181

[Peyton-Jones 87]
Simon L. Peyton-Jones
The Implementation of Functional Programming Languages
(Prentice-Hall International Series in Computer Science)
Prentice Hall, 1987

[Pratt 84]
Terrence W. Pratt
Programming Languages: Design and Implementation [2nd Edition]
Prentice Hall, Inc., 1984

[Rees & Clinger 86]
Jonathan Rees and William Clinger (Editors)
Revised^3 Report on the Algorithmic Language SCHEME
MIT/AI/Memo 848a or Indiana University CS/TR 174 or
ACM SIGPLAN Notices, Vol.21 No.12, Dec 1986

[Reynolds 74]
John C. Reynolds
Towards a Theory of Type Structure
In Proc. Colloque sur la Programmation,
Lecture Notes in Computer Science No.19, Springer-Verlag 1974, pp.408–425

[Rosser 82]
J. Barkley Rosser
Highlights of the History of the λ-Calculus
In Proc. ACM Symposium on the History of Programming Languages, 1982, pp.216–225

[Sebesta 89]
Robert W. Sebesta
Concepts of Programming Languages
Benjamin/Cummings Publishing Company, Inc., 1989

[Snyder 86]
Alan Snyder
Encapsulation and Inheritance in Object-Oriented Programming Languages
In Proc. ACM OOPSLA '86, Sept 1986, pp.38–45

[Steele 85/90]
Guy L. Steele Jr.
C MM N LISP*: The Language*
Digital Press, 1985, Second Edition 1990

[Tarjan 83]
Robert Endré Tarjan
Data Structures and Network Algorithms
Chapter 2 (Disjoint Sets)
Society for Industrial and Applied Mathematics, 1983

[Ward & Halstead 89]
Stephen A. Ward and Robert H. Halstead, Jr.
Computation Structures
The MIT Electrical Engineering and Computer Science Series
The MIT Press, 1990

[Wirth 71]
Niklaus Wirth
The Programming Language PASCAL
In Acta Informatica, Vol.1, 1971, pp.35–63

[van Wijngaarden *et al.* 76]
Aad van Wijngaarden *et al.*
Revised Report on the Algorithmic Language ALG L 68
Springer-Verlag, 1976, pp.103–107

[X3J13 88]
Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon
C MM N LISP *Object System Specification X3J13 Document 88-002R*
In ACM SIGPLAN Notices, Vol.23 Special Issue, Sep 1988
Also, Chapter 28 of [Steele 85/90, Second Edition]

[XLRG 81]
The Xerox Learning Research Group
The SMALLTALK-80 *System*
BYTE, Vol.6 No.8, Aug 1981, pp.36–48