Raw, Pickled and Textured Static Side-Effect Analysis in FX-87

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

Michael R. Blair† [ziggy@lcs.mit.edu]

†MIT Laboratory for Computer Science

Abstract

Designers of optimizing compilers have long been stymied by first-class procedures where recursion and side-effects are permitted. So severe is the problem that most research in this area has focused on languages which do not support these features, such as F RTRAN.

Lucassen and Gifford introduced polymorphic effect systems to alleviate this problem. However, their approach has been shown to suffer a couple shortcomings [Harrison 89, p.364]. I remedy these problems by developing a more aggressive polymorphic region-based effect system based on an intuitive notion of subtypes-as-subsets but augmented with several views of the semantics of region-based effects.

A kernel language, $*$-$\mathcal{TREK}$, with a polymorphic region-based effect system is introduced. Using it as a framework, four different views of region-based effects are developed and used to attack four aspects of providing an effective and efficient region-based effect system. Specifically, region-based effects are viewed as annotating run-time properties of programs ranging from: storage cell usage, to process synchronization requirements, to manipulation of aggregate data, to patterns of data sharing and interference. These views are used in turn to safely permit: disregarding of (many) unobservable local side effects, subtyping in passing procedural arguments, assignment in the face of subtyping coercions, and compile-time coordination of concurrency.

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

Static analysis is the life-blood of optimizing compilation, but ensuring the soundness of many optimizations pivots on the availability of both dataflow and control flow graphs. These graphs have been shown to be expensive to extract precisely, particularly when we allow procedures as arguments where recursion and side-effects are permitted.

Polymorphic Effect Systems were introduced by Lucassen and Gifford in an effort to alleviate this predicament somewhat [Gifford & Lucassen 86] [Lucassen & Gifford 88]. Such systems employ type annotations of procedure arguments as carriers of the desired dependence information. Specifically, region-based polymorphic effect systems encode, within the type of procedures, the effect incurred on the store when the procedure is invoked. In addition, aggregate data are annotated with the region of the store in which their components have been allocated. Moreover, parameters and local variables, insofar as they denote binding cells, are annotated with the region in which their bindings occur. Immutable data is characterized by being allocated in the distinguished *Immutable Region*.

Within this framework, a process called Effect Masking enables the compiler to remove from a procedure's effect annotation those local effects which are externally non-observable when the procedure is invoked. For example, local variables can be allocated and mutated for internal bookkeeping but when this behavior is not externally observable then the procedure is still purely functional from the viewpoint of any potential invoker. This permits a mixture of imperative and functional programming styles without loss of evaluation concurrency.

Unfortunately, the polymorphic type and effect system introduced in their language FX-87 [Gifford *et al.* 87] reflected many design trade-offs [Lucassen 87]. Often, precision was traded for terseness or purported simplicity. In this paper, we recoup the precision lost there by separating the intuitive notion of subtypes-as-subsets from the soundness and safety issues of incorporating region-based effects into the system. Further, by distinguishing four distinct viewpoints of region-based effect annotations we capitalize on their rich information content to permit a more precise Effect Masking rule and to accept a larger class of programs with more opportunities for parallelism than does FX-87. In the process, a deeper understanding of the strengths and limitations of polymorphic region-based effect systems should develop.

The remainder of this paper is structured as follows. Section 2 presents a brief overview of region-based effect systems in general. Section 3 introduces the kernel language $*$-$\mathcal{TREK}$ which sports a polymorphic region-based effect system based strongly on FX-87 [Gifford *et al.* 87]. In Section 4, I distinguish a handful of syntactic classes of region descriptions in order to define an effective and efficient Effect Masking Rule based on a storage-cell model of region-based effects. In Section 5, a process-synchronization model of effects is introduced and used as a basis for safely permitting subtyping when passing procedures as parameters. In Section 6, a collecting-semantics model of effects is used as a basis for permitting subtyping of arguments when passing aggregate data to non-functional procedures. In Section 7, a sharing/interference model of regions and effects is developed and used as a basis for compile-time regulation of concurrency when applying polymorphic procedures. Finally, Section 8 summarizes our results and Section 9 compares our approach to related work in the field.

2 Overview of Effect Systems and Stratified Typing

With effect systems, we are interested in statically analyzing more than just the types of expressions: we are interested as well in their run-time side effects. It is through this static side-effect analysis that the soundness of compiler optimizations such as those mentioned in the preceding section can be proven. Effect systems extend existing type declarations and inference rules so as to document the side-effects of expressions as well as their types. This is done by extending the type system into a more general *kind* system, requiring declarations of kind `EFFECT` as well as those of kind `TYPE`. Effects, in turn, document the sets of locations in the store that they affect. In region-based effect systems this is achieved by conceptually coloring the store into distinctly colored `REGION`s, thus further extending the kind system to include declarations of kind `REGION`. In such a system, every well-formed expression will have both a type and an effect. The type describes the value computed and the effect describes accesses and alterations incurred on the store.

In such a system, procedures will have, embedded within their types, the declaration of their latent effects.[1] Any expression in which the procedure appears in operator position of an application will be charged by the type checker for realizing this latent effect. Thus, procedure types are used as carriers for latent effects. In this way, *Descriptions* (generalized declarations) are used as a means of checking the types and effects of value expressions while using the meta-language of kinds to check the well-formedness of these descriptions. Similarly, aggregate data structures will contain, embedded within their types, the regions in which their components are allocated.[2]

The next section details the syntax and much of the type/effect checking rules of a kernel region-based effect system which I call $*$-$\mathcal{TREK}$.[3]

3 $*$-$\mathcal{TREK}$: A Kernel Region-Based Effect System

3.1 The $*$-$\mathcal{TREK}$ Language Grammar (Extended BNF)

3.1.1 [Kinds]

$$\kappa \ ::= \ \mathtt{REGION} \mid \mathtt{EFFECT} \mid \mathtt{TYPE}$$

3.1.2 [Descriptions]

$$\delta \ ::= \ \rho \mid \epsilon \mid \tau$$

$$\rho \ ::= \ \mathtt{@=} \mid \mathtt{@}\ \mathit{identifier} \mid (\mathtt{RUNION}\ \rho\ \rho\ \ldots) \mid \mathit{param}$$

$$\begin{array}{l} \epsilon \ ::= \ \mathtt{PURE} \mid (\mathtt{READ}\ \rho) \mid (\mathtt{ALLOC}\ \rho) \mid (\mathtt{WRITE}\ \rho) \\ \quad \mid (\mathtt{MAXEFF}\ \epsilon\ \ldots) \mid \mathit{param} \end{array}$$

$$\tau \ ::= \ \tau_{\mathrm{base}} \mid \tau_{\mathrm{aggr}} \mid \tau_{\mathrm{recr}} \mid \tau_{\mathrm{poly}} \mid \mathit{param}$$

$$\begin{array}{lll} \tau_{\mathrm{base}} & ::= & \mathtt{VOID} \mid \mathtt{UNIT} \mid \mathtt{BOOLE} \mid \mathtt{INT} \mid \mathtt{SYMBOL} \mid \mathtt{STRING} \\ \tau_{\mathrm{aggr}} & ::= & (\mathtt{SUBR}\ \epsilon\ (\tau\ \ldots)\ \tau) \\ & & \mid (\mathtt{RECORDOF}\ ((\mathit{tag}\ \tau)\ \ldots)\ \rho) \\ & & \mid (\ \ \mathtt{ONEOF}\ ((\mathit{tag}\ \tau)\ \ldots)\ \rho) \\ \tau_{\mathrm{recr}} & ::= & (\mathtt{RECTYPE}\ \mathit{param}\ \tau) \\ \tau_{\mathrm{poly}} & ::= & (\mathtt{POLY}\ ((\mathit{param}\ \kappa)\ \ldots)\ \tau) \end{array}$$

$$\begin{array}{rcl} \mathit{param} & ::= & \mathit{identifier} \\ \mathit{tag} & ::= & \mathit{identifier} \end{array}$$

3.1.3 [Values]

For clarity in some of the productions below, I have included superscripts which are intended to be vaguely suggestive of type annotations.

$$\upsilon \ ::= \ \upsilon_{\mathrm{base}} \mid \upsilon_{\mathrm{aggr}} \mid \upsilon_{\mathrm{abst}} \mid \upsilon_{\mathrm{ctrl}} \mid \upsilon_{\mathrm{decl}}$$

$$\begin{array}{lll} \upsilon_{\mathrm{base}} & ::= & \mathtt{\#U} \mid \mathtt{\#T} \mid \mathtt{\#F} \mid \mathbb{Z} \mid \text{'}\ \mathit{identifier} \mid \text{"}\ \mathit{char}\ldots\text{"} \\ \upsilon_{\mathrm{aggr}} & ::= & \upsilon_{\mathrm{param}} \mid \upsilon_{\mathrm{recrd}} \mid \upsilon_{\mathrm{oneof}} \end{array}$$

$$\begin{array}{lll} \upsilon_{\mathrm{param}} & ::= & \mathit{param} \mid (\mathtt{SET!}\ \mathit{param}\ \upsilon) \\ \upsilon_{\mathrm{recrd}} & ::= & (\mathtt{RECORD}\ ((\mathit{tag}\ \upsilon)\ \ldots)\ \rho) \\ & & \mid (\mathtt{RECORD\text{-}SET!}\ \upsilon^{\mathrm{recrd}}\ \mathit{tag}\ \upsilon) \\ & & \mid (\mathtt{SELECT}\ \mathit{tag}\ \upsilon^{\mathrm{recrd}}) \\ \upsilon_{\mathrm{oneof}} & ::= & (\mathtt{ONE}\ \tau^{\mathrm{oneof}}\ \mathit{tag}\ \upsilon) \\ & & \mid (\mathtt{ONE\text{-}SET!}\ \upsilon^{\mathrm{oneof}}\ \mathit{tag}\ \upsilon) \\ & & \mid (\mathtt{TAGCASE}\ \mathit{param}^{\mathrm{oneof}} \\ & & \quad (\mathit{tag}\ \upsilon)\ldots\ (\mathtt{ELSE}\ \upsilon)) \end{array}$$

$$\upsilon_{\mathrm{abst}} \ ::= \ \upsilon_{\lambda\ \mathrm{abst}} \mid \upsilon_{\pi\ \mathrm{abst}}$$

$$\begin{array}{lll} \upsilon_{\lambda\ \mathrm{abst}} & ::= & (\upsilon^{\lambda\ \mathrm{abst}}\ \upsilon\ldots) \\ & & \mid (\ \lambda\ \ ((\mathit{param}\ \tau\ \rho)\ \ldots)\ \upsilon) \\ & & \mid (\ \mathtt{LETREC}\ ((\mathit{param}\ \upsilon^{[P]\lambda}\ \rho)\ \ldots)\ \upsilon) \\ \upsilon_{\pi\ \mathrm{abst}} & ::= & (\mathtt{P}\lambda\ \ ((\mathit{param}\ \kappa)\ \ldots)\ \upsilon) \\ & & \mid (\mathtt{PROJ}\ \upsilon^{\pi\ \mathrm{abst}}\ \delta\ldots) \end{array}$$

$$\upsilon_{\mathrm{ctrl}} \ ::= \ \upsilon_{\mathrm{brnch}} \mid \upsilon_{\mathrm{sequ}}$$

$$\begin{array}{lll} \upsilon_{\mathrm{brnch}} & ::= & (\mathtt{IF}\ \upsilon^{\mathrm{Boole}}\ \upsilon\ \upsilon) \\ \upsilon_{\mathrm{sequ}} & ::= & (\mathtt{BEGIN}\ \upsilon\ \upsilon\ldots) \end{array}$$

$$\upsilon_{\mathrm{decl}} \ ::= \ (\mathtt{THE}\ \epsilon\ \tau\ \upsilon)$$

$$\begin{array}{rcl} \mathit{param} & ::= & \mathit{identifier} \\ \mathit{tag} & ::= & \mathit{identifier} \end{array}$$

3.2 Description Equivalence

3.2.1 [Basic Equivalence Axioms]

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

$$\begin{array}{rcl} & & \delta \equiv \delta \\ \delta \equiv \delta' \ \wedge \ \delta' \equiv \delta'' & \Longrightarrow & \delta \equiv \delta'' \\ \delta \equiv \delta' & \Longrightarrow & \delta' \equiv \delta \\ \delta \equiv \delta' & \Longrightarrow & \delta'' \equiv \delta''\{\delta'/\delta\} \end{array}$$

...where $\delta''\{\delta'/\delta\}$ denotes δ'' with the substitution of δ' for any or all occurrences of δ within δ''. This last rule (substitutivity) makes our otherwise mere equivalence relation

[1] Which may be the effect PURE, indicating that there is no observable side effect.

[2] There exists a distinguished *Immutable Region* as well as an axiom which states that any region containing the Immutable Region is itself immutable.

[3] Read as "Star Trek". The "trek" is an acronym of Type, Region, Effect, and Kind. The "*" indicates that this is a kernel design for a continuum of larger languages could be built; it also suggests that a continuum of increasingly aggressive systems are possible.

be a congruence relation. Nonetheless, it is traditional to call type congruence relations "type equivalence" relations, hence I use the symbol $\equiv$ rather than the more precise symbol $\cong$.

3.2.2 [Region Equivalence]

A region can be viewed as a set of region names, possibly parameterized via a `POLY` abstraction.

$$
\begin{array}{rcl}
(\texttt{RUNION}\ \ \rho\ \ ) & \equiv & \rho \\
(\texttt{RUNION}\ \rho\ \rho\ ...) & \equiv & (\texttt{RUNION}\ \rho ...) \\
(\texttt{RUNION}\ (\texttt{RUNION}\ \rho_1 ... \rho_n)\ \rho_{n+1} ...) & \equiv & (\texttt{RUNION}\ \rho_1 ... \rho_n\ \rho_{n+1} ...) \\
(\texttt{RUNION}\ \rho_1 ... \rho_n) & \equiv & (\texttt{RUNION}\ \rho_{\pi(1)} ... \rho_{\pi(n)})
\end{array}
$$

...for any permutation function π of the positive integers $1 \dots n$.

3.2.3 [Effect Equivalence]

`MAXEFF` expresses a set of effects ϵ, so axioms analogous to the `RUNION` axioms above apply: substitute `MAXEFF` for `RUNION` and ϵ for ρ. Effects, however, have additional structure:

$$
\begin{array}{rcl}
(\texttt{MAXEFF}) & \equiv & \texttt{PURE} \\
(\texttt{MAXEFF}\ \texttt{PURE}\ \epsilon) & \equiv & (\texttt{MAXEFF}\ \epsilon) \\
\\
(\texttt{MAXEFF}\ (\texttt{READ}\ \ \rho)\ (\texttt{READ}\ \ \rho')) & \equiv & (\texttt{READ}\ \ (\texttt{RUNION}\ \rho\ \rho')) \\
(\texttt{MAXEFF}\ (\texttt{ALLOC}\ \rho)\ (\texttt{ALLOC}\ \rho')) & \equiv & (\texttt{ALLOC}\ (\texttt{RUNION}\ \rho\ \rho')) \\
(\texttt{MAXEFF}\ (\texttt{WRITE}\ \rho)\ (\texttt{WRITE}\ \rho')) & \equiv & (\texttt{WRITE}\ (\texttt{RUNION}\ \rho\ \rho'))
\end{array}
$$

Thus, a `MAXEFF` can be viewed as a set of `READ`s, `ALLOC`s, `WRITE`s (each possibly parameterized) and `POLY`-bound effect parameters.

Moreover, a special restriction holds for `WRITE` effects on the distinguished immutable region `@=` (read "at equal"):

$$
\begin{array}{rcl}
(\texttt{WRITE}\ \texttt{@=}) & \equiv & \texttt{error}_{\text{effect}} \\
(\texttt{MAXEFF}\ \epsilon\ \texttt{error}_{\text{effect}}) & \equiv & \texttt{error}_{\text{effect}}
\end{array}
$$

If we view an effect as describing a set of expressions which will incur (any subeffect of) that effect on the store when evaluated, then $\texttt{error}_{\text{effect}}$ describes *all* expressions, including those which incur illegal effects. Thus, one can view the forbidden effect `(WRITE @=)` as the top of the effect lattice ($\top_{\text{effect}}$) since no valid expression will incur this effect. Similarly, effect `PURE` is the bottom element ($\perp_{\text{effect}}$).

3.2.4 [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 `ONEOF`s and `RECORDOF`s are not order-specific:

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

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

3.2.5 [RECTYPE Equivalence]

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

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

3.2.6 [Higher Order Description Equivalence]

The following constitutes the familiar α-renaming rule of the λ-calculus [Barendregt 84] [Stoy 77] [Church 51]:

$$
\frac{\forall \texttt{dvar}'\ .\ \texttt{dvar}' \ \notin\ \texttt{FDV}(\tau_{\text{body}}) \cup \{\texttt{dvar}...\}}{(\texttt{POLY}\ ((\texttt{dvar}\ \kappa)...)\ \tau_{\text{body}})\ \equiv\ (\texttt{POLY}\ ((\texttt{dvar}'\ \kappa)...)\ \tau_{\text{body}}[\texttt{dvar}'/\texttt{dvar}][...])}
$$

...where $\delta\,[\delta'/i][...]$ denotes δ with the simultaneous substitution of the δ'... for the free occurrences of the i... within δ, α-renaming as appropriate to avoid capture, with "free occurrences" defined in the obvious way [Lucassen 87, §3.3.5, p.31f]. Likewise, `FDV` is the Free Description Variables function.

3.2.7 [Completeness of the Equivalence Axioms]

Description expressions that 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

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.

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

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 inequation. For instance:

$$(\texttt{SUBR}\ \epsilon_\downarrow\ (\tau_{1\uparrow}\ ...\ \tau_{n\uparrow})\ \tau_{\text{rtn}\downarrow})\ \ \sqsubseteq\ \ (\texttt{SUBR}\ \epsilon_\uparrow\ (\tau_{1\downarrow}\ ...\ \tau_{n\downarrow})\ \tau_{\text{rtn}\uparrow})$$

is shorthand for the more verbose:

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

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 with some effect on the store, then the set of procedures which are defined on a possibly larger domain but which always map to a possibly smaller range with a possibly smaller set of side effects on the store 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 [Region Inclusion]

$$\begin{array}{rcl} (\texttt{RUNION}\ \rho_{1\downarrow}\ ...\ \rho_{n\downarrow}) & \sqsubseteq & (\texttt{RUNION}\ \rho_{1\uparrow}\ ...\ \rho_{n\uparrow}) \\ \rho_{\downarrow} & \sqsubseteq & (\texttt{RUNION}\ \rho_{\uparrow}\ \rho') \end{array}$$

3.3.4 [Effect Inclusion]

$$\begin{array}{rcl} (\texttt{READ}\ \ \rho_{\downarrow}) & \sqsubseteq & (\texttt{READ}\ \ \rho_{\uparrow}) \\ (\texttt{ALLOC}\ \rho_{\downarrow}) & \sqsubseteq & (\texttt{ALLOC}\ \rho_{\uparrow}) \\ (\texttt{WRITE}\ \rho_{\downarrow}) & \sqsubseteq & (\texttt{WRITE}\ \rho_{\uparrow}) \end{array}$$

$$\begin{array}{rcl} (\texttt{MAXEFF}\ \epsilon_{1\downarrow}\ ...\ \epsilon_{n\downarrow}) & \sqsubseteq & (\texttt{MAXEFF}\ \epsilon_{1\uparrow}\ ...\ \epsilon_{n\uparrow}) \\ \epsilon_{\downarrow} & \sqsubseteq & (\texttt{MAXEFF}\ \epsilon_{\uparrow}\ \epsilon') \end{array}$$

3.3.5 [Type Inclusion]

$$\texttt{VOID} \sqsubseteq \tau \quad \forall\tau$$

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

$$\begin{array}{ll} & (\texttt{ONEOF}\quad ((\texttt{tag}_1\ \tau_{1\downarrow})...(\texttt{tag}_n\ \tau_{n\downarrow}))\ \rho_{\downarrow}) \\ \sqsubseteq & (\texttt{ONEOF}\quad ((\texttt{tag}_1\ \tau_{1\uparrow})...(\texttt{tag}_n\ \tau_{n\uparrow}) \\ & \qquad\qquad (\texttt{tag}'\ \tau'\ )...(\texttt{tag}''\ \tau''\ ))\ \rho_{\uparrow}) \end{array}$$

$$\begin{array}{ll} & (\texttt{RECORDOF}\ ((\texttt{tag}_1\ \tau_{1\downarrow})...(\texttt{tag}_n\ \tau_{n\downarrow}) \\ & \qquad\qquad (\texttt{tag}'\ \tau'\ )...(\texttt{tag}''\ \tau''\ ))\ \rho_{\downarrow}) \\ \sqsubseteq & (\texttt{RECORDOF}\ ((\texttt{tag}_1\ \tau_{1\uparrow})...(\texttt{tag}_n\ \tau_{n\uparrow}))\ \rho_{\uparrow}) \end{array}$$

This defines VOID to be the bottom of the TYPE partial order induced by $\sqsubseteq$. There are no values of type VOID: it is the type of the result of invoking the ERROR procedure.

The asymmetry in the RECORDOF and ONEOF rules deserves comment. RECORDOF forms describe vectors of data which 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 which 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.

3.3.6 [Higher Order Description Inclusion]

$$(\texttt{POLY}\ ((\texttt{dvar}\ \kappa)...)\ \tau_{\mathrm{body}\downarrow}) \sqsubseteq (\texttt{POLY}\ ((\texttt{dvar}\ \kappa)...)\ \tau_{\mathrm{body}\uparrow})$$

3.3.7 [RECTYPE Inclusion]

As with equivalence, a RECTYPE form is included in another description iff its (possibly) infinitely unwound body is "point-wise" 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 $\mathcal{A}_{\sqsubseteq}$ denotes a set of assumed inclusion constraints on RECTYPE parameters and where $\mathcal{A}_{\sqsubseteq} \vdash_{\sqsubseteq}$ R means that assuming $\mathcal{A}_{\sqsubseteq}$ one can prove using the $\sqsubseteq$ rules that relation R is satisfied.

$$\frac{\mathcal{A}_{\sqsubseteq} \cup \{(\texttt{RECTYPE t}\ \tau_{\downarrow}) \sqsubseteq \tau_{\uparrow}\} \quad \vdash_{\sqsubseteq} \quad \tau_{\downarrow}[(\texttt{RECTYPE t}\ \tau_{\downarrow})/\texttt{t}] \sqsubseteq \tau_{\uparrow}}{\mathcal{A}_{\sqsubseteq} \quad \vdash_{\sqsubseteq} \quad (\texttt{RECTYPE t}\ \tau_{\downarrow}) \sqsubseteq \tau_{\uparrow}}$$

$$\frac{\mathcal{A}_{\sqsubseteq} \cup \{\tau_{\downarrow} \sqsubseteq (\texttt{RECTYPE t}\ \tau_{\uparrow})\} \quad \vdash_{\sqsubseteq} \quad \tau_{\downarrow} \sqsubseteq \tau_{\uparrow}[(\texttt{RECTYPE t}\ \tau_{\uparrow})/\texttt{t}]}{\mathcal{A}_{\sqsubseteq} \quad \vdash_{\sqsubseteq} \quad \tau_{\downarrow} \sqsubseteq (\texttt{RECTYPE t}\ \tau_{\uparrow})}$$

Notice that until now our inclusion rules have all, in effect, been of the form: $\{\} \ \vdash_{\sqsubseteq}\ \delta_{\downarrow} \sqsubseteq \delta_{\uparrow}$.

3.4 Expression Type and Effect Inference

The following inference rules use a value/description/kind environment (VDKenv) which maps value variables to their value, type, and the region in which they are bound, and which maps description variables to their descriptions and kinds. If a value variable is an uninstantiated parameter, then this maps only to its type (there being no value). Similarly, uninstantiated description parameters are mapped only to a kind.

Following are a selected few type/effect rules intended to convey some interesting features of type and effect checking in our language. The rules for RECORDs and ONEOFs are not shown, nor is the rule for LETREC, which defines mutually recursive (possibly polymorphic) procedures, or the THE rule, which explicitly declares a value to be an instance of a supertype and/or supereffect.

In the following, "`:`" means "has type"; "`!`" means "has effect"; "`::`" means "has kind"; VDKenv$[x\!:\!\tau@\rho][x\!::\!\kappa]$ denotes environment extension; $\delta[\delta'/\texttt{dvar}]$ denotes δ with the substitution of δ' for all free occurrences of dvar within δ; FDV(x) selects the free description variables of a description or value expression; and FV(v) selects the free value variables of a value expression, where *free variables* is defined in the obvious way. The subsequent section clarifies some of the less traditional conditions stipulated in some of these rules and mentions a couple subtle issues which these rules suppress.

3.4.1 [Value Variable Type & Effect]

$$\mathrm{VDKenv}[\ \texttt{var}\!:\!\tau@\rho] \ \vdash\ \ \texttt{var} : \tau\ \texttt{!}\ (\texttt{READ}\ \rho)$$

3.4.2 [SET! Type & Effect]

$$\frac{\begin{array}{rcl}\{\} & \vdash_{\sqsubseteq} & \tau_{\text{val}} \sqsubseteq \tau_{\text{var}} \\ \text{VDKenv} & \vdash & \upsilon : \tau_{\text{val}} \ \texttt{!}\ \epsilon_{\text{val}}\end{array}}{\text{VDKenv}[\ \texttt{var}{:}\tau_{\text{var}}\texttt{@}\rho_{\text{var}}] \vdash \begin{array}{l}(\texttt{SET!}\ \ \texttt{var}\ \upsilon) \\ : \texttt{UNIT} \\ \texttt{!}\ (\texttt{MAXEFF}\ \epsilon_{\text{val}}\ \ (\texttt{WRITE}\ \rho_{\text{var}}))\end{array}}$$

3.4.3 [IF Type & Effect]

$$\frac{\begin{array}{lclcll}\text{VDKenv} & \vdash & \upsilon_{\text{tst}} & : & \texttt{BOOLE} & \texttt{!}\ \epsilon_{\text{tst}} \\ \text{VDKenv} & \vdash & \upsilon_{\text{con}} & : & \tau_{\text{con}} & \texttt{!}\ \epsilon_{\text{con}} \\ \text{VDKenv} & \vdash & \upsilon_{\text{alt}} & : & \tau_{\text{alt}} & \texttt{!}\ \epsilon_{\text{alt}}\end{array}}{\text{VDKenv} \vdash \begin{array}{l}(\texttt{IF}\ \upsilon_{\text{tst}}\ \upsilon_{\text{con}}\ \upsilon_{\text{alt}}\ ) \\ : (\texttt{MAXTYPE}\ \tau_{\text{con}}\ \ \tau_{\text{alt}}) \\ \texttt{!}\ (\texttt{MAXEFF}\ \epsilon_{\text{tst}}\ \ (\texttt{EITHER}\ \epsilon_{\text{con}}\ \ \epsilon_{\text{alt}}))\end{array}}$$

Notice that rather than compute a least upper bound (LUBTYPE),[4] I tentatively prefer to follow [Gifford *et al.* 87] by settling for a lesser mechanism, $(\texttt{MAXTYPE}\ \tau_{\text{con}}\ \tau_{\text{alt}})$, where:

$$(\texttt{MAXTYPE}\ \tau_1\ \tau_2) = \begin{cases} \tau_1 & \text{if } \tau_2 \sqsubseteq \tau_1 \\ \tau_2 & \text{if } \tau_1 \sqsubseteq \tau_2 \\ \text{error} & \text{otherwise.} \end{cases}$$

When the MAXTYPE is error, the IF rule would be said to fail, thus the IF form in question would be said to fail to type check.

The peculiar structure of the overall effect of the IF form will be justified in Section 6. For now, suffice it to say that in reading the effect from left to right, one could interpret this to indicate that ϵ_{tst} is always incurred, followed by either ϵ_{con} or ϵ_{alt}. In this way, effects can be arranged to constitute a *collecting semantics* [Nielson 87/88] of the effects incurred by value expressions. The EITHER effect constructor is algebraically equivalent to MAXEFF but it will be treated specially when doing ϵ-modulated inclusion.

3.4.4 [BEGIN Type & Effect]

$$\frac{\text{VDKenv} \vdash \upsilon_i : \tau_i \ \texttt{!}\ \epsilon_i}{\text{VDKenv} \vdash (\texttt{BEGIN}\ \upsilon_1\ \ldots\ \upsilon_n) : \tau_n \ \texttt{!}\ (\texttt{MAXEFF}\ \epsilon_1\ \ldots\ \epsilon_n)}$$

3.4.5 [λ Type & Effect]

$$\frac{\text{VDKenv}[\ \texttt{var}_i{:}\tau_i\texttt{@}\rho_i] \vdash \upsilon : \tau \ \texttt{!}\ \epsilon}{\text{VDKenv} \vdash \begin{array}{l}(\lambda\ ((\ \texttt{var}_1\ \tau_1\ \rho_1)\ldots(\ \texttt{var}_n\ \tau_n\ \rho_n))\ \upsilon) \\ : (\texttt{SUBR}\ \epsilon_\alpha\ \ (\tau_1\ \ldots\ \tau_n)\ \tau)\ \texttt{!}\ \ \texttt{PURE}\end{array}}$$

$$\text{where } \epsilon_\alpha = \begin{cases} (\texttt{MAXEFF}\ (\texttt{ALLOC}\ (\texttt{RUNION}\ \rho_1\ \ldots\ \rho_n))\ \epsilon) & n \geq 1 \\ \epsilon & n = 0 \end{cases}$$

As we shall discuss in Section 4, ϵ_α will, in fact, be subjected to Effect Masking to mask any non-observable effects occurring during the execution of the procedure.

3.4.6 [Application Type & Effect]

$$\frac{\begin{array}{lclcll}\text{VDKenv} & \vdash & \upsilon_\psi & : & (\texttt{SUBR}\ \epsilon_\psi\ \ (\tau_{\chi 1}\ \ldots\ \tau_{\chi n})\ \tau_{\text{rtn}}) & \texttt{!}\ \ \epsilon_0 \\ \text{VDKenv} & \vdash & \upsilon_i & : & \tau_{\alpha i} & \texttt{!}\ \ \epsilon_i \\ \{\} & \vdash_{\sqsubseteq} & \tau_{\alpha i} & \sqsubseteq_{\epsilon_\psi} & \tau_{\chi i} & \end{array}}{\text{VDKenv} \vdash \begin{array}{l}(\upsilon_\psi\ \upsilon_1\ \ldots\ \upsilon_n) : \tau_{\text{rtn}} \\ \texttt{!}\ (\texttt{MAXEFF}\ (\texttt{PI}\ \epsilon_0\ \ldots\ \epsilon_n)\ \epsilon_\psi)\end{array}}$$

...for any permutation function PI which maps a variable number of effects to a MAXEFF. Were we to assure a specific order of evaluation of arguments, we would choose PI accordingly. Since I prefer to leave the order of evaluation of arguments unspecified, we will leave PI as a pseudonym for MAXEFF.

Notice that this rule permits actual arguments' types to be in inclusion relation with the types expected by the operator, where this inclusion is modulated by the latent effect ϵ_ψ of the operator. This notion of *ϵ-modulated type inclusion* is developed in Section 6.

In this regard, my development differs from that of FX-87. I have chosen to base inclusion on the intuitive notion that types denote sets of values so subtypes should naturally denote subsets. When faced with the safety issue of allowing subtyping on arguments to applications which incur WRITE effects, I prefer to address that concern squarely in the application type checking rule by introducing ϵ-modulated type inclusion there.

By contrast, the FX-87 designers chose to make the application type checking rule straightforward (as above) while allowing this safety concern to spill over into the inclusion axioms for aggregate types. In that context, the intuitive notion of subtype is burdened with the additional overhead of denoting a subset of values *which can safely replace their supertypes in all contexts*, consequently requiring each aggregate description inclusion axiom to be conditionalized on the mutability of the region component of the supertype. To repeat, this issue is addressed in detail in Section 6.

3.4.7 [Pλ Type & Effect]

$$\forall\ \texttt{var} \in \text{FV}(\upsilon)\ \neg\exists_{1 \leq i \leq n}\ .\ \texttt{dvar}_i \in \text{FDV}(\text{VDKenv}(\ \texttt{var})) :$$

$$\frac{\text{VDKenv}[\texttt{dvar}_i :: \kappa_i] \vdash \upsilon : \tau \ \texttt{!}\ \texttt{PURE}}{\text{VDKenv} \vdash \begin{array}{l}(\texttt{P}\lambda\ \ ((\texttt{dvar}_1\ \kappa_1)\ldots(\texttt{dvar}_n\ \kappa_n))\ \upsilon) \\ : (\texttt{POLY}\ ((\texttt{dvar}_1\ \kappa_1)\ldots(\texttt{dvar}_n\ \kappa_n))\ \tau) \\ \texttt{!}\ \texttt{PURE}\end{array}}$$

Consider the quantified conditional in the first clause of this rule. This stipulation avoids capture of description variables appearing within the types of the free variables of a polymorphic expression. This concern was discovered by McCracken and was dubbed the "safe" condition on polymorphic types [McCracken 79, p.20f], hereafter referred to in this paper as the *McCracken Criterion.* Operationally, forward substituting locally defined description variables and systematically α-renaming all locally defined description parameters to avoid collision with globally defined description variables achieves the same effect.

Note further the stipulation that the body of the Pλ be PURE. This ensures that projection will incur no run-time expense: projection is entirely static in polymorphic region-based effect systems. To do otherwise would require an embedded (latent) effect in POLY (like in SUBR) which would be realized upon projecting with PROJ (analogous to the way the application rule realizes latent SUBR effects).[5]

3.4.8 [PROJ Type & Effect]

$$\frac{\begin{array}{lcl}\text{VDKenv} & \vdash & \upsilon : (\texttt{POLY}\ ((\texttt{dvar}_1\ \kappa_1)\ldots(\texttt{dvar}_n\ \kappa_n))\ \tau)\ \texttt{!}\ \epsilon \\ \text{VDKenv} & \vdash & \delta_i :: \kappa_i\end{array}}{\text{VDKenv} \vdash (\texttt{PROJ}\ \upsilon\ \delta_1\ \ldots\ \delta_n) : \tau[\delta_i/\texttt{dvar}_i]\ \texttt{!}\ \epsilon}$$

[4] Indeed, in our system the lub does not always exist. For example, in our system INT and BOOLE are incomparable.

[5] In fact, MFX did precisely that [Lucassen 87, Chapt.3] but the designers of FX-87 chose to adopt the more conservative approach proposed in [*Ibid.*, Chapt.7].

The projection rule, as stated above, does not allow parallelization of polymorphic procedures about their polymorphization parameters. This is because in a polymorphic effect system, the compiler does compile-time parallelism scheduling/synchronization of a polymorphic procedure's body upon evaluating its definition, so it must anticipate all possible interference patterns which projection may introduce. Thus, since no information can be assumed about how the polymorphized parameters will eventually be projected, if the compiler actually used the rule as stated above then it would have to compile a Pλ with the assumption of maximal interference/sharing regarding these parameters.

To combat this sequentializing pessimistic assumption, the FX-87 Reference Manual stipulates that it be considered a static error for any projection to take place which might actually introduce interference/sharing patterns about these polymorphization parameters [Gifford *et al.* 87, p.40]. This allows the compiler to recover parallelism in the compiled Pλ body by entitling it to assume instead that there need be *no* sequentialization with respect to the parameters, but this comes at the cost of a static restriction.

This restriction can be relaxed: it is excessively conservative to forbid *all* interference/sharing with respect to these parameters when all that is necessary is to forbid any *new* interference/sharing. Consequently, the *Region Anti-Aliasing Rule* on which this restriction is based can be made more effective than that definition in the FX-87 manual. This issue of interference/sharing is explored in detail in Section 7.

3.4.9 [Effect Masking]

The final subtlety lurking within the type and effect inference rules is the notion that side effects to the store which are locally visible within a block induce local synchronization on the processes which execute the procedure. However, side effects which are not observable outside of some lexical contour should not induce synchronization outside of that contour. The rule which approximates this goal is called the *Effect Masking Rule* [Gifford *et al.* 87, p.30]. This rule was formalized and its soundness was proposed by Lucassen in [Gifford *et al.* 87, Appendix B: FX Semantics]. This issue is addressed in Section 4, where the groundwork established in [Gifford *et al.* 87] is refined.

3.5 Soundness of the Type and Effect Inference Rules

Lucassen [Lucassen 87] conjectured the soundness of the Type and Effect Inference Rules for FX-87. A similar conjecture holds for $*$-$\mathcal{TREK}$. This means, in short, that the type and effect of an expression, as inferred by these rules, is a conservative estimate of the actual type and effect of the expression. He noted that, as a consequence, the compiler writer can use these rules as a basis for various optimizations of the object code. These include automatic storage reclamation (*e.g.,* via stack allocation) of variables and formals whose values are not returned out of the scope of their definition (*i.e.,* those of local extent), memoization (software caching) of pure expressions, common subexpression elimination of pure expressions, dead-code elimination of expressions whose value is not used and which do not mutate the store, code motion within side-effect synchronization contours, and concurrent evaluation non-interfering expressions [Aho & Ullman 86]. Again, the inference soundness further entitles us to avoid type checking at run time, along with its associated potential time and space inefficiencies.

4 Raw Effects for Effect Masking

In this section, we explore how effect annotations can be used to minimize the latent effects embedded within SUBR types. Our approach is to use a generate-and-test paradigm: we compute the effects of a procedure's body using the type and effect inference rules of the previous section, afterward applying the Effect Masking Rule to the computed effect. It is this masked effect which is embedded in the SUBR type. The rule is based on a storage-cell model of effects— that is, we view READ, ALLOC, WRITE effects (RAW, or "raw", effects) as denoting manipulations of memory cells in a mutable store. We now proceed to derive an Effect Masking Rule.

4.1 Observability of READ and WRITE Effects

We now derive a READ and WRITE effect masking rule by considering the following (contrived but nonetheless illustrative) sample program:

```
(DEFINE global-mut ((PROJ VECTOR @GMV) #F #F))

(DEFINE global-reader  (λ () (VECTOR-REF  global-mut 0)))
(DEFINE global-writer! (λ ((b BOOLE))
                          (VECTOR-SET! global-mut 1 b)))
(DEFINE global-string "Down" @GSTRING)
(DEFINE global-symbol 'BEAM)

(DEFINE xport (λ ((whom SYMBOL)(where STRING))
  (LET* ((local-mut ((PROJ VECTOR @LMV) 'SPOCK 'SCOTT 'KIRK))
         (local-reader (λ () (VECTOR-REF local-mut 1))))
    (global-reader)
    (global-writer! #T)
    (SET! global-string where)
    (VECTOR-SET! local-mut 0 whom)
    (VECTOR global-symbol
            (VECTOR-REF  local-mut 0)
            global-string
            (local-reader)
            (VECTOR-REF global-mut 1)))))

(xport 'ME "Up")
```

Ignoring for now all ALLOC effects,[6] let's consider step by step what observable READ and WRITE effects the call to xport may have when evaluated.

First, global-reader is invoked, which will incur a READ effect on the region in which the components of global-mut were allocated (*viz.,* @GMV); however, this READ is not observable since it was performed by a piece of non-returning code— that is, the expression does not have an observable WRITE effect and its return value is not used so this expression can be dead-code eliminated at compile time. Were this not the case, the READ could not be masked. Notice, by the way, that this maskable (READ @GMV) effect appears in the type of global-reader.

Second, global-writer! is invoked. Again, the result of this procedure invocation is not used, but unlike the call to global-reader, global-writer! incurs a WRITE effect (on @GMV). This WRITE effect cannot be masked since it will be externally visible to processes performing READ effects to @GMV. Notice, by the way, that this (WRITE @GMV) appears in the type of global-writer!.

[6] In fact, xport has no externally observable ALLOC effects when invoked.

Next, a SET! is performed on `global-string`, which incurs a (WRITE @GSTRING) effect. This cannot be masked since it is observable by any other process which may reference `global-string`.

Next, a WRITE effect is incurred on the locally defined `local-mut`. Since this vector is never exposed to external processes, this WRITE is unobservable outside the LET expression; however, within the LET expression this WRITE *is* observable so it should not be masked until outside the scope of the LET binding.

Lastly, we return a VECTOR but only after having evaluated its arguments in an unspecified order. Imagining, for the sake of argument, that the arguments are evaluated left-to-right this time, the reference to `global-symbol` will be PURE since it is not defined to be a mutable variable.

Next, the VECTOR-REF to `local-mut` will incur a READ effect on @LMV because this is where the variable's binding was ALLOCd, but it will be masked outside the scope of the LET since `local-mut` is not externally accessible.

By contrast, the reference to `global-string` will incur a READ on @GSTRING which cannot be masked since external WRITEs to it must be properly synchronized with respect to this READ to preserve the sequential semantics of the language. For example, if the above call to `export` appeared in a BEGIN code block in which a preceding expression incurred a WRITE to @GSTRING, then a data dependency would exist between the two expressions so a synchronization point must be imposed to maintain the sequential semantics of the code block.

Next, the invocation of `local-reader` will incur a READ effect on @LMV which will be unobservable outside the scope of the LET, thus it can be masked.

And finally, the VECTOR-REF to `global-mut` will incur an unmaskable READ effect since this is a mutable externally visible datum.

4.2 Classes of Region Occurrences

The above discussion encourages us to distinguish four distinct classes of region occurrences, which are enumerated below. Specifically, our first two considerations focused on two of these classes: those regions which appear within the types of free SUBR variables either within 1) READ or 2) WRITE components. More generally, we should consider those components of any free variable whose type may contain a SUBR type, for instance a REF to a SUBR or even a PURE SUBR which returns a SUBR with latent READ/WRITE effects.[7] The third consideration involved 3) those regions in which free variables have been bound— that is, not where the binding's components have been allocated, but specifically where the variable binding itself has been allocated. For example, to define a global mutable variable we must specify where that variable is to allocate its binding so that subsequent SET! expressions can mutate it. This is contrasted with the fourth consideration, which isolated interest on 4) those region components in which free mutable structure components were allocated— that is, the regions of the components of the bindings, not the regions of the bindings themselves. The final considerations further involved READ effects over these region occurrence classes.

Computing precisely the set of locations observably read and written in the canonicalization of an expression is, in general, undecidable. Even settling for a minimally conservative decidable approximation would require a complex and potentially expensive non-standard abstract interpretation of value expressions [Weihl 80]. As an alternative, effect masking allows us to perform a simpler, primarily syntactic flow-insensitive analysis of an expression to compute a conservative approximation of the same information but to do so based on regions rather than on locations *per se*. If the programmer is careful to allocate distinguished locations in distinct regions, then this effect masking should produce precisely the same information as a more complex semantic analysis.[8]

Returning to the derivation of a READ and WRITE effect masking rule, we are now equipped to formally define selectors for the classes of region occurrence described earlier. First, we define selectors ρ_{R} and ρ_{W} (read "region READ occurrences" and "region WRITE occurrences") for region occurrences within a description, δ, as follows:

$\rho_{\mathrm{R}}(\delta)$ — regions appearing within READ effects of SUBR latent effects and return types in δ or those regions appearing within region components of returned structure types in δ

$\rho_{\mathrm{W}}(\delta)$ — regions appearing within WRITE effects of SUBR latent effects and return types in δ or those regions appearing within region components of returned mutable structure types in δ

Next, we define FV_ρ and FV_τ which select the binding regions and types of the free variables of a value expression.

$\mathrm{FV}_\rho(\nu, \mathrm{env})$ = regions where free vars of ν are bound in env
$\mathrm{FV}_\tau(\nu, \mathrm{env})$ = types of free vars of ν as per env bindings

Now, we define the region occurrence classes desired for masking as follows (the boxed numbers relate these terms to the region occurrence classes as enumerated at the beginning of this section):

$$\mathcal{R}_\rho(\nu, \mathrm{env}) = \overbrace{\{\bigcup_i \rho_{\mathrm{R}}(\tau_i) \mid \tau_i \in \mathrm{FV}_\tau(\nu, \mathrm{env})\}}^{\boxed{1}\ \&\ \boxed{4}} \cup \overbrace{\mathrm{FV}_\rho(\nu, \mathrm{env})}^{\boxed{3}}$$

$$\mathcal{W}_\rho(\nu, \mathrm{env}) = \underbrace{\{\bigcup_i \rho_{\mathrm{W}}(\tau_i) \mid \tau_i \in \mathrm{FV}_\tau(\nu, \mathrm{env})\}}_{\boxed{2}\ \&\ \boxed{4}} \cup \underbrace{\mathrm{FV}_\rho(\nu, \mathrm{env})}_{\boxed{3}}$$

We read $\mathcal{R}_\rho$ as "observable READ regions" and $\mathcal{W}_\rho$ as "observable WRITE regions".

Claim 4.1 *$\mathcal{R}_\rho$ and $\mathcal{W}_\rho$ compute conservative estimates of the regions observably read and written, respectively, in the canonicalization of state $\langle \nu, \mathrm{env}, \sigma \rangle$ for any choice of store σ.*

4.3 The READ and WRITE Effect Masking Rule

Using the notation of the preceding section, we may now formalize our earlier discussion of observable READ and WRITE effects by the following READ and WRITE Effect Masking Operators and the READ and WRITE Effect Masking Rule followed by a soundness claim relating effect masking over the type and effect inference rules to value canonicalization.

[7] Notice that we need not consider the argument types in SUBRs. This point will be discussed more carefully when considering ALLOC masking below.

[8] Unfortunately, it is not always possible to allocate distinct locations in distinct regions. For example, looping constructs like DO must allocate locals in the same regions with each iteration.

Definition 4.1 [READ and WRITE Effect Masking Operators]

- $\mathcal{R}_{\text{unmaskable}}(\rho_r, \nu, \text{env}) \;=\; \rho_r \sqcap \text{inj}\rho(\mathcal{R}_\rho(\nu, \text{env}))$
- $\mathcal{W}_{\text{unmaskable}}(\rho_w, \nu, \text{env}) \;=\; \rho_w \sqcap \text{inj}\rho(\mathcal{W}_\rho(\nu, \text{env}))$

Moreover, dead-code READ*s may be ignored when computing* $\mathcal{R}_\rho(\nu, \text{env})$ *(and a warning should be issued if indeed any are discovered).*

Here and throughout, injρ maps a set of (closed) region descriptions to the RUNION over those region descriptions.[9]

Here and throughout, $\sqcap$ denotes the "meet" operator defined as the greatest lower bound (glb) with respect to description inclusion ($\sqsubseteq$).[10] Intuitively, the READ and WRITE masking operators define those regions in ρ_r (resp., ρ_w) which may not be disregarded when considering the observable READ (resp., WRITE) effects incurred in the canonicalization of ν in environment env and any store σ. By contraposition, therefore, those regions not selected by these operators can be disregarded.

Lemma 4.1 [Soundness of READ and WRITE Effect Masking Operators]
The set of observable READ *(resp.,* WRITE*) effects incurred by the canonicalization of a state is a subset of the* READ *(resp.,* WRITE*) effects remaining after applying the* READ *(resp.,* WRITE*) effect masking operator to the region of the* READ *(resp.,* WRITE*) component of the effect computed by the type and effect inference rules.*

We can therefore safely incorporate these effect masking operators into the type and effect inference system via the following Effect Masking Rule:

Definition 4.2 [READ and WRITE Effect Masking Rule]

$$\frac{\langle \text{env}, \sigma\rangle \;\vdash\; \nu \;!\; (\texttt{MAXEFF}\ (\texttt{READ}\ \rho_r)(\texttt{WRITE}\ \rho_w)(\texttt{ALLOC}\ \rho_a))}{\langle \text{env}, \sigma\rangle \;\vdash\; \nu \;!\; (\texttt{MAXEFF}\ (\texttt{READ}\ \mathcal{R}_{\text{unmaskable}}(\rho_r, \nu, \text{env}))\ (\texttt{WRITE}\ \mathcal{W}_{\text{unmaskable}}(\rho_w, \nu, \text{env}))\ (\texttt{ALLOC}\ \rho_a))}$$

Lemma 4.2 [Soundness of Effect Inference with READ and WRITE Masking]
Extending the system of type and effect inference rules of Section 3.4 to incorporate the READ *and* WRITE *Effect Masking Rule preserves the soundness of the type and effect inference rules proved by Lucassen [Lucassen 87].*

This means, in short, that the type and effect of an expression, as inferred by the type and effect inference rules with READ and WRITE effect masking, is a conservative estimate of the actual type and effect of the expression. As a consequence, the compiler writer can use these rules as a basis for various optimizations of the object code. As mentioned in Section 3.5, this masking expands the set of expressions for which the compiler can soundly employ memoization (software caching), common subexpression elimination, code motion within side-effect boundaries, and parallelization of non-interfering expressions. Naturally, the objective of any implementation of the resulting inference system is to compute the least type and effect possible with respect to the inclusion axioms.

[9] By convention, inj$\rho(\{\}) = \perp_{\text{REGION}}$, the bottom of the region partial order, where by definition (READ $\perp_{\text{REGION}}$) = (WRITE $\perp_{\text{REGION}}$) = (ALLOC $\perp_{\text{REGION}}$) = PURE. Note that $\perp_{\text{REGION}}$ is not equivalent to @= since (WRITE @=) is forbidden.

[10] We will use this only over region descriptions, for which they are always defined given that regions denote sets and given our newly introduced null-region $\perp_{\text{REGION}}$.

4.4 Observability of ALLOC Effects

In short, ALLOC effects are visible only when the allocated memory is "exposed". Allocation is exposed in only one of three ways: a mutable aggregate is allocated (*e.g.,* via CONS) and is permitted to escape from the lexical context in which it was created either 1) by being directly returned within the return value, or 2) by being stored into an externally defined datum; or 3) a mutable local variable is allocated and it is closed over by one or more locally defined procedures which collectively both read and write the local and which are permitted to escape from the lexical context in which they were created.[11]

An (admittedly contrived) program fragment which exemplifies all these cases follows:

```
(DEFINE global-mut ((PROJ VECTOR @HAT)    ; a catty pair...
                    ((PROJ CONS @BLUE) 'DOLPHIN 'WHALE)))
(DEFINE seuss-itate (λ ()
  (LET (( red-fish 'HERRING @RED)          ; a mutable local-var
        (blue-fish ((PROJ CONS @BLUE ) 'SCROD           'COD))
        (ges&h     ((PROJ CONS @GREEN) 'CHICKEN-EMBRYOS 'N1DZE)))
    (VECTOR-SET! global-mut 0 blue-fish)
    (LET ((reader   (λ () red-fish))
          (mutator! (λ ((new-fish SYMBOL))
                      (SET! red-fish new-fish))))
      (CONS (VECTOR reader mutator!) ges&h)))))

(DEFINE seuss-pool (seuss-itate))
```

Note that we are not concerned with the allocation of immutable values (such as the local variables blue-fish and ges&h) but rather with the allocation of mutable values only (such as the mutable pairs to which blue-fish and ges&h are bound, and the mutable local red-fish over which reader and mutator! are closed).

When invoked, seuss-itate will have observable ALLOC effects on the @RED, @BLUE, and @GREEN regions. Let's consider each distinct allocation in turn.

First, consider what would happen if only reader were returned (via the returned vector) but not mutator!. In that case, two separate evaluations of the LET expression would produce two distinct reader procedures which are nonetheless operationally indistinguishable. This is because no means will have been provided whereby either instances of the locals over which the readers were closed could be altered. Conversely, if only mutator! were returned but not reader, two distinct procedures would be produced that mutate a closed local which no process (not even themselves!) would be able to read. Thus, in both cases the allocation to @RED could be made to immutable storage instead since in the one case no mutation would be feasible while in the second the mutation would be effectively unobservable.

Of course, to allow ALLOC masking by virtue of providing only a reader or only a mutator to a local is likely to be a reflection of a programmer error, so it would not be unreasonable for an implementation to signal a warning in such situations to encourage the programmer to re-seuss-itate the program.

With this first maskability issue in mind, consider the type of the value returned:

[11] Not unexpectedly, these are all instances of the more general notion that whenever the extent of a mutable locally allocated datum exceeds its scope, that allocation is potentially observable.

```
(PAIROF (VECTOROF (SUBR (READ  @RED)  ()   INT)
                  (SUBR (WRITE @RED) (INT) INT)
            @=)
        (PAIROF SYMBOL SYMBOL @GREEN)
    @=)
```

The important detail here is that within this return type there appears both a READ and a WRITE effect on the region in which the mutable local will be allocated (*viz.,* @RED). We conclude from this that ALLOC effects to regions which appear in *both* READ and WRITE regions within the latent effect (or return type) of a returned SUBR component may not be masked.

Notice one subtlety above in the treatment of SUBR types. Were we to return only a reader, but one which accepts an argument which is itself a SUBR with a WRITE effect, this argument type of the reader would not indicate an ability to mutate the local which we have allocated. Thus, for purposes of seeking out READ and WRITE effects of returned SUBRs, we always completely ignore the arguments' types, looking only at the latent effect component and the return type since only these components reveal if the SUBR returned will expose the local storage or will itself return a SUBR which will expose the local storage. More generally, when discussing any region components of a type, we will heretofore take this to mean that the argument-type components within SUBR types are to be ignored.

Next, imagine if the storage of blue-fish into global-mut via the call to VECTOR-SET! had not been performed. Then the allocation of the blue-fish pair components in @BLUE would be unobservable. The interesting detail to note here is that whenever a global variable (or, more generally, a free variable) appears within an expression wherein some newly allocated structure is stored into that variable, the type of the variable's binding (in this example, (PAIROF SYMBOL SYMBOL @BLUE)) must mention the region in which the new structure's components were allocated in order for the mutation to be permitted. Thus, if the programmer takes care to allocate local structure components in regions which do not collide with the region components appearing in the types of free variables bound to mutable structure, then we may conclude with no loss of precision that ALLOC effects to regions which do collide in this manner may not be masked. This is a modest, natural discipline to impose on the programmer who is interested in precise masking. In essence, it encourages the programmer to use regions to encode patterns of potential sharing.

Finally, the locally allocated mutable pair ges&h was returned as a component of the pair returned by the LET expression. Notice that, therefore, the region in which ges&h's mutable components were allocated (*viz.,* @GREEN) will appear in the region component of a mutable structure type within the return type (*viz.,* (PAIROF SYMBOL SYMBOL @GREEN)). Therefore, we conclude that ALLOC effects to regions appearing within mutable structure components of the result type of an expression may not be masked.

As with detecting observable READ and WRITE effects, a non-standard abstract interpretation of value expressions could be defined to precisely detect observable ALLOC effects, but again this would be complicated and potentially expensive. Again, therefore, we develop a more simple, primarily syntactic analysis of an expression to compute a conservative approximation of the same information but to do so based on regions rather than on locations *per se*. Specifically, using this notation, we can relate the semantic and syntactic ALLOC properties of expressions as follows:

Claim 4.2 *By merely analyzing the type of a canonicalized value, we can discover all regions of the store which can be observably read and written via manipulation of the value given a store.*

Returning now to the derivation of an ALLOC effect masking rule, we are equipped to formally define the "observable ALLOC regions" occurrence class as:

$$\begin{array}{rcl} \mathcal{A}_\rho(\nu, \mathrm{env}, \sigma, \tau, \rho_w) & = & \rho_{\mathrm{R\&W}}(\tau) \ \cup \ \mathcal{R}\&\mathcal{W}_\rho(\nu, \mathrm{env}, \rho_w) \\ \text{where} \qquad \rho_{\mathrm{R\&W}}(\delta) & = & \{\rho \mid \{\} \vdash_{\sqsubseteq} \rho \sqsubseteq \mathrm{inj}\rho(\rho_R(\delta)) \sqcap \mathrm{inj}\rho(\rho_W(\delta))\} \end{array}$$

$$\mathcal{R}\&\mathcal{W}_\rho(\nu, \mathrm{env}, \rho_w) = \begin{cases} \text{if} & \mathcal{W}_{\mathrm{unmaskable}}(\rho_w, \nu, \mathrm{env}) = \{\} \\ \text{then} & \{\} \\ \text{else} & \{\rho \mid \{\} \vdash_{\sqsubseteq} \rho \sqsubseteq \mathrm{inj}\rho(\mathcal{R}_\rho(\nu, \mathrm{env})) \sqcap \\ & \qquad \mathrm{inj}\rho(\mathcal{W}_\rho(\nu, \mathrm{env}))\} \end{cases}$$

Naturally, we read $\rho_{\mathrm{R\&W}}$ as "region READ&WRITE occurrences" and $\mathcal{R}\&\mathcal{W}_\rho$ as "observable READ&WRITE regions".

Notice that this suggests that ALLOC effect masking presumes READ and WRITE effect masking. Specifically, the conditional within the definition of $\mathcal{R}\&\mathcal{W}_\rho(\nu, \mathrm{env}, \rho_w)$ reflects our concern with exposing a locally allocated location by writing it into a free variable in a manner which would make the allocation observable by virtue of there existing both read and write access to the variable's value components. Clearly, if no writes occur during the canonicalization of ν,[12] then this is a mute concern. We will see later that this can be relaxed ever when writes do occur.

4.5 The ALLOC Effect Masking Rule

Using the notation of the preceding section, we may now summarize the discussion of observable ALLOC effects by the following ALLOC Effect Masking Rule followed by a soundness claim relating effect masking over the type and effect inference rules to value reduction.

Definition 4.3 **[ALLOC Effect Masking Operator]**

- $\mathcal{A}_{\mathrm{unmaskable}}(\rho_a, \nu, \mathrm{env}, \sigma, \tau, \rho_w) \ = \ \rho_a \sqcap \mathrm{inj}\rho(\mathcal{A}_\rho(\nu, \mathrm{env}, \sigma, \tau, \rho_w))$

Intuitively, this operator defines those regions in ρ_a which may not be disregarded when considering the ALLOC effects incurred in the canonicalization of ν in environment env and store σ where, in addition, τ and the regions in ρ_w reflect potential points of exposure of allocate effects. Conversely, those regions not selected by this operator can be disregarded.

Lemma 4.3 **[Soundness of ALLOC Effect Masking Operator]**
The set of observable ALLOC *effects incurred by the canonicalization of a state is a subset of the* ALLOC *effects remaining after applying the* ALLOC *effect masking operator to the region of the* ALLOC *effect computed by the type and effect inference rules, using the type of the return value and the region of the* WRITE *effect computed by the inference rules as indications of when local allocations may have been exposed.*

We therefore incorporate this final effect masking operator into the type and effect inference system via the following

[12] *I.e.*, $\rho_w = \bot_{\mathrm{REGION}}$

full Effect Masking Rule:

Definition 4.4 [Full $*$-$\mathcal{TREK}$ Effect Masking Rule]

$$\frac{\langle \text{env}, \sigma \rangle \quad \vdash \quad \nu : \tau \ \texttt{! (MAXEFF (READ } \rho_r\texttt{)(WRITE } \rho_w\texttt{)(ALLOC } \rho_a\texttt{))}}{\begin{array}{lll} \langle \text{env}, \sigma \rangle & \vdash & \nu : \tau \\ & & \texttt{! (MAXEFF (READ } \mathcal{R}_{\text{unmaskable}}(\rho_r, \nu, \text{env})\texttt{)} \\ & & \qquad\qquad \texttt{(WRITE } \mathcal{W}_{\text{unmaskable}}(\rho_w, \nu, \text{env})\texttt{)} \\ & & \qquad\qquad \texttt{(ALLOC } \mathcal{A}_{\text{unmaskable}}(\rho_a, \nu, \text{env}, \sigma, \tau, \rho_w)\texttt{))} \end{array}}$$

Lemma 4.4 [Soundness of Effect Inference with Full $*$-$\mathcal{TREK}$ Effect Masking]
Extending the type and effect inference system of Section 3.4 to incorporate the Full $*$-$\mathcal{TREK}$ Effect Masking Rule preserves the soundness of the type and effect inference rules proved by Lucassen [Lucassen 87].

This means, in short, that the type and effect of an expression, as inferred by the type and effect inference rules with full $*$-$\mathcal{TREK}$ effect masking, is a conservative estimate of the actual type and effect of the expression. As a consequence, the compiler writer can use these rules as a basis for various optimizations of the object code. As mentioned in Section 3.5, the newly incorporated `ALLOC` masking expands the set of expressions for which the compiler can soundly employ memoization (software caching), stack allocation (and hence automatic garbage collection) of local non-observably allocated temporary values, and common subexpression elimination.

4.6 Summary of Effect Masking

To summarize, this section derived a precise Effect Masking Rule defined in terms of the distinct syntactic classes of region occurrences which one naturally considers when trying to decide if an effect is observable at some point within an expression. This rule is, presumably, more straightforward than the terse rule presented in the FX-87 Reference Manual [Gifford *et al.* 87, p.30] and it yields more precise results in the sense that it masks effects that the "official" manual masking rule could not.

Further, the plausibility of efficiently implementing this more powerful rule was presented. Moreover, since the newly derived rule filters out more irrelevant information than does the former rule— for example, by altogether ignoring argument types within `SUBR` types— and since it compares only those segments of the collected information which are relevant— for instance, it considers only $\rho_{\text{R\&W}}$ in `ALLOC` masking rather than both the complete sets ρ_{R} and ρ_{W}, and the `READ` and `WRITE` masking operators consider only one of ρ_{R} or ρ_{W}, respectively— the new masking rule derived here is likely to execute faster than the prescribed rule from the manual. This is clear if one views the effect masking process as using a generate-and-test paradigm where local effects are generated which may later be masked based on the success/failure of the masking test. The improved rule generates fewer initial local effects and it can be accelerated by appropriate data structures to separate the independent mask testing concerns.

5 Pickled Effects for Procedural Arguments

The previous section viewed `READ`/`ALLOC`/`WRITE`effect annotations ("raw effects") as describing an expression's manipulation of storage cells. This is a fruitful viewpoint for performing effect masking, but it is not the only semantics one can ascribe to effects. In this section, we adopt a view of effects as describing how expressions with given effects must be synchronized in a concurrent implementation to maintain the deterministic sequential semantics of the expressions. We also consider the possibility of memoizing the results of pure processes. This view of effects shifts perspective to a model based on `PRODUCE`, `CONSUME`, `LOCK` (PCL, or "pickled", effects) constraints on the concurrent processes spawned by invoking procedures with annotated latent effects.

5.1 The Semantics of Effect Inclusion

The intent of our type inclusion axioms is to permit a programmer to pass an argument whose type is a subtype of the expected type declared in the parameter list of the procedure being invoked. For non-procedural arguments, this is justified by viewing subtypes as subsets: if we expect a datum whose value can range over a set of values then it is acceptable to pass a datum whose value ranges over a subset of those expected.

For procedure arguments, this issue is more subtle. Here we accept a procedure which can be invoked safely anywhere on the expected domain, hence any procedure with a larger domain than that expected. Also, we accept any procedure which, when invoked, will yield a value in the range (codomain) of the expected type, hence any procedure with a small range.

But what of the latent effect of a procedure? Certainly, it is safe to pass a procedure whose latent effect is a subset of the effects expected since this indicates that it will perform only a subset of the expected manipulations of the store. However, effect inclusion based solely on raw effects is restrictive. Consider how the compiler coordinates concurrency within the body of a procedure which accepts a procedural parameter.

If the procedure has latent effect `PURE` then invocations of the procedure can run concurrently with the evaluation of any other expression. Moreover, calls to the procedure can be memoized so repeated calls can be common-subexpression eliminated (CSE).

If the procedure parameter has an `ALLOC` effect on some region ρ then calls to it cannot be memoized since it will, in general, return a freshly allocated cell upon each call. Thus, calls to it cannot be CSEd. Other expressions, however, can proceed concurrently since they cannot interfere with the allocation. This is because no other expression will have a handle on the newly allocated cell until the allocater returns.

If, on the other hand, the procedure parameter incurs a `READ` effect on some region ρ, then memoization and CSE are not safe. Concurrent evaluation, however, is still possible with the following limitation: other expressions which incur a `WRITE` effect on ρ must be synchronized to maintain the sequential semantics of the language. That is, evaluation of writers to ρ which are to be evaluated before our reader under the sequential semantics must return finish evaluating before initiating a call to our reader. Similarly, writers to be evaluated subsequent to our reader call must wait for our reader to return. However, other readers of ρ can be evaluated concurrently with calls to our reader, so some degree of concurrency is still safe.

Finally, if the procedure parameter has a `WRITE` effect on some region ρ, then all other expressions which incur either `READ` or `WRITE` effects on ρ must be sequentialized with respect to our writer. Moreover, calls to writers cannot be memoized or CSEd.[13]

Notice, now, that the restrictions on writers are strictly more limiting that the restriction on readers: readers to ρ cannot evaluate concurrently with our writer to ρ. Likewise, the restriction on readers is strictly more than that on writers, and the restriction on allocaters strictly exceeds that on PURE functional procedures. This restriction total order is depicted by the following:

$$\texttt{PURE} \prec \texttt{ALLOC} \prec \texttt{READ} \prec \texttt{WRITE}$$

5.2 Process-Synchronization Model of Effects

This looks peculiar since a procedure which WRITEs a cell behaves characteristically differently from a procedure which READs a cell. To clarify that our interest is in the synchronization of the processes spawned by a procedure and not on the action *per se* of the procedure, we can imagine our raw effects to be renamed according to the following schema:

$$\begin{array}{rcl} \texttt{ALLOC} & \leadsto & \texttt{PRODUCE} \\ \texttt{READ} & \leadsto & \texttt{CONSUME} \\ \texttt{WRITE} & \leadsto & \texttt{LOCK} \end{array}$$

Our synchronization ordering then becomes:

$$\texttt{PURE} \prec \texttt{PRODUCE} \prec \texttt{CONSUME} \prec \texttt{LOCK}$$

This PCL view of effects I call "pickled" effects.

Based on this restriction ordering, it is safe to invoke a procedure with a less restrictive effect in a place where a procedure with a more restrictive effect was anticipated. For instance, it is safe to pass a consumer on ρ where a locker on ρ is expected. Moreover, if a procedure incurs two different effects on the same region, for instance being both a producer and a consumer on ρ, then this is no more restrictive than having just the more restrictive of the effects, in this case the consumer effect.

5.3 Pickled Effect Axioms

These more tolerant constraints on the types of procedures than can be safely passed are formalized by the following axioms on pickled effects:

5.3.1 Pickled Effect Equivalence

$$\begin{array}{rcl} \texttt{(ALLOC}\ \rho\texttt{)} & \equiv & \texttt{(PRODUCE}\ \rho\texttt{)} \\ \texttt{(READ}\ \rho\texttt{)} & \equiv & \texttt{(CONSUME}\ \rho\texttt{)} \\ \texttt{(WRITE}\ \rho\texttt{)} & \equiv & \texttt{(LOCK}\ \rho\texttt{)} \\ \\ \texttt{(MAXEFF (LOCK}\ \rho\texttt{) (CONSUME}\ \rho\texttt{))} & \equiv & \texttt{(LOCK}\ \rho\texttt{)} \\ \texttt{(MAXEFF (LOCK}\ \rho\texttt{) (PRODUCE}\ \rho\texttt{))} & \equiv & \texttt{(LOCK}\ \rho\texttt{)} \\ \texttt{(MAXEFF (CONSUME}\ \rho\texttt{) (PRODUCE}\ \rho\texttt{))} & \equiv & \texttt{(CONSUME}\ \rho\texttt{)} \end{array}$$

[13] Well, unless two writer calls appear with provably identical arguments (textually identical PURE expressions) and no other writes to intersecting regions are incurred by textually intermediate expressions.

5.3.2 Pickled Effect Inclusion

$$\begin{array}{rcl} \texttt{(CONSUME}\ \rho_\downarrow\texttt{)} & \sqsubseteq & \texttt{(LOCK}\ \rho_\uparrow\texttt{)} \\ \texttt{(PRODUCE}\ \rho_\downarrow\texttt{)} & \sqsubseteq & \texttt{(LOCK}\ \rho_\uparrow\texttt{)} \\ \texttt{(PRODUCE}\ \rho_\downarrow\texttt{)} & \sqsubseteq & \texttt{(CONSUME}\ \rho_\uparrow\texttt{)} \end{array}$$

5.4 Summary of Pickled Effects

Insofar as our type and effect inference rules are defined in terms of $\sqsubseteq$, incorporating these additional axioms into $\equiv$ and $\sqsubseteq$ is transparent. In fact, since we can pass procedures as components of say RECORDOF forms, we automatically permit more flexible inclusion beyond the simple case of passing procedure arguments. The IF rule for example is more forgiving now.

Notice, however, that our type and effect rules produce only raw effects, not pickled effects. This is by design since the pickled effects are desirable for performing inclusion tests but the raw effects are what is needed to do sound effect masking. Were we to omit lesser effects from the latent effects of procedures then, for instance, our test for $\rho_{\mathrm{R\&W}}$ would have to change to a test of just ρ_{W} since a procedure which both reads and writes some region would perhaps appear as one which only writes the region. This would force less precise effect masking.

In short, raw effects are needed for doing precise effect masking but pickled effects are preferred when testing type inclusion. This reflects the difference in emphasis between storage cell issues and processes synchronization issues.

6 ϵ-Modulated Type Inclusion for Aggregate Arguments

We now address the safety issue raised by permitting subtyping when passing aggregate data to procedures which mutate their arguments. As alluded to in Section 3.4.3, this will involve viewing effect annotations in yet another way. This time, we will view effects as constituting a *collecting semantics* [Nielson 87/88] in the sense that they describe a pattern of storage manipulation which an expression may incur. For effects inferred using the type and effect inference rules, these effects will also reflect a temporal sequencing of these manipulations. In this sense, effects will be considered to denote a traversal path through the store taken when a procedure is invoked. User supplied effects will not, in general, encode this valuable information, as we shall discuss. First, let's clarify this safety concern.

6.1 The Problem

The imperative minded reader will have, by this point, become concerned with the soundness of performing assignment to components of structured data such as RECORDOF and ONEOF data. The concern is that procedure application has heretofore been presented as if it permitted the arguments to have types which are subtypes of the formals' declared types. If we view subtyping intuitively as set inclusion on those values approximated by the types, then component mutation of structured arguments raises a subtle soundness issue. Specifically, consider the following figure, where we imagine (REF τ ρ) to be an abbreviation for (RECORDOF ((val τ)) ρ):

According to the type inclusion axioms [§ 3.3.5], REF [$\downarrow$] is a subtype of REF [$\uparrow$] since SUB is a subtype of SUPER. Thus,

```
[↓] (REF x @!)          ----------          (REF x @!) [↑]
        |          /      SUPER      \            |
        |       /         ------        \       /
          \    |       /   SUB      \       |    /
           \__|_____|____ \/         |   \/__|___/
             |      |    / \        | / \   |
             |       \          /       |
               \        ------       /
                  \                /
                    ----------
```

Figure 1: Structured Data Subtyping

by the application type checking rule [§ 3.4], we should be permitted to pass a REF [↓] to any procedure which expects an argument of type REF [↑]. Now consider that in general there can be many pointers to a value of type REF [↓]. These other aliases for the same REF will be used by other procedures to manipulate the component of this REF. Type soundness would be lost were it not guaranteed that any mutation to the component of a REF will always be equivalent to or a subtype of the current component's type. This is as prescribed by the type checking rule for SET. For instance, it would be safe for us to SET the component of a REF [↑] to be a value of type SUB, but it would be unsound to SET the component of a REF [↓] to be a value of type SUPER. Unfortunately, if we pass a REF [↓] to a procedure which expects a REF [↑], and that procedure mutates the cell of the REF passed to it so that the REF afterwards contains a new value of type SUPER, then that procedure will have violated type soundness.

6.2 A Solution

The way to remedy this potential problem is to use a more constrained notion of type inclusion when passing structured data as arguments. We need to consider the writable regions in the type of the operator being applied to the structured datum and to modulate our argument type inclusion tests according to the intersection of the operator type's ρ_{W}s and the arguments types' read/write and mutable structure regions, $\rho_{\mathrm{R\&W}}$. This new modulated inclusion we denote as $\sqsubseteq_\epsilon$ (read "is included in, modulo ϵ").

6.2.1 ϵ-Modulated Application Rule

$$
\frac{\begin{array}{rlcll}
\text{VDKenv} & \vdash & \upsilon_\psi & : & (\texttt{SUBR}\ \epsilon_\psi\ (\tau_{\chi 1}\ \ldots\ \tau_{\chi n})\ \tau_{\mathrm{rtn}})\ \ \texttt{!}\ \ \epsilon_0 \\
\text{VDKenv} & \vdash & \upsilon_i & : & \tau_{\alpha i}\ \ \texttt{!}\ \ \epsilon_i \\
\{\} & \vdash_{\sqsubseteq} & \tau_{\alpha i} & \sqsubseteq_{\epsilon_\psi} & \tau_{\chi i}
\end{array}}{\text{VDKenv}\ \vdash\ (\upsilon_\psi\ \upsilon_1\ \ldots\ \upsilon_n)\ :\ \tau_{\mathrm{rtn}}\ \ \texttt{!}\ (\texttt{MAXEFF}\ \epsilon_{\pi(0)}\ \ldots\ \epsilon_{\pi(n)}\ \epsilon_\psi)}
$$

6.2.2 ϵ-Modulated Inclusion Axioms

This new ϵ-modulated inclusion ($\sqsubseteq_\epsilon$) relation is defined inductively over the structured types. For the base types, τ_{base}, $\sqsubseteq_\epsilon$ is just $\sqsubseteq$.

$$
\frac{\begin{array}{rcll}
\epsilon_\downarrow & \sqsubseteq & \epsilon_\uparrow & \\
\tau_{i\downarrow} & \sqsubseteq & \tau_{i\uparrow} & \forall i\ .\ 1 \leq i \leq n \\
\tau_{\mathrm{rtn}\downarrow} & \sqsubseteq_{\epsilon \succeq \epsilon_\uparrow} & \tau_{\mathrm{rtn}\uparrow} &
\end{array}}{(\texttt{SUBR}\ \epsilon_\downarrow\ (\tau_{1\uparrow}\ \ldots\ \tau_{n\uparrow})\ \tau_{\mathrm{rtn}\downarrow})\ \sqsubseteq_\epsilon\ (\texttt{SUBR}\ \epsilon_\uparrow\ (\tau_{1\downarrow}\ \ldots\ \tau_{n\downarrow})\ \tau_{\mathrm{rtn}\uparrow})}
$$

$$
\frac{\begin{array}{rlcl}
 & \rho_\downarrow \sqsubseteq \rho_\uparrow & & \\
\text{If} & \texttt{@=} \sqsubseteq \rho_\uparrow & \vee & \neg\exists\ \rho \in \rho_{\mathrm{W}}(\epsilon)\ .\ \rho \sqsubseteq \rho_\uparrow \\
\text{Then} & \tau_{i\downarrow} \sqsubseteq_{\epsilon \succeq (read\ \rho_\uparrow)} \tau_{i\uparrow} & & \\
\text{Else} & \tau_{i\downarrow} \equiv \tau_{i\uparrow} & &
\end{array}}{\begin{array}{rl}
 & (\texttt{ONEOF}\ ((\texttt{tag}_1\ \tau_{1\downarrow}) \ldots (\texttt{tag}_n\ \tau_{n\downarrow}))\ \rho_\downarrow) \\
\sqsubseteq_\epsilon & (\texttt{ONEOF}\ ((\texttt{tag}_1\ \tau_{1\uparrow}) \ldots (\texttt{tag}_n\ \tau_{n\uparrow}) \\
 & \qquad (\texttt{tag}'\ \tau'\ ) \ldots (\texttt{tag}''\ \tau''\ ))\ \rho_\uparrow)
\end{array}}
$$

$$
\frac{\begin{array}{rlcl}
 & \rho_\downarrow \sqsubseteq \rho_\uparrow & & \\
\text{If} & \texttt{@=} \sqsubseteq \rho_\uparrow & \vee & \neg\exists\ \rho \in \rho_{\mathrm{W}}(\epsilon)\ .\ \rho \sqsubseteq \rho_\uparrow \\
\text{Then} & \tau_{i\downarrow} \sqsubseteq_{\epsilon \succeq (read\ \rho_\uparrow)} \tau_{i\uparrow} & & \\
\text{Else} & \tau_{i\downarrow} \equiv \tau_{i\uparrow} & \wedge & m = 0
\end{array}}{\begin{array}{rl}
 & (\texttt{RECORDOF}\ ((\texttt{tag}_1\ \tau_{1\downarrow}) \ldots (\texttt{tag}_n\ \tau_{n\downarrow}) \\
 & \qquad (\texttt{tag}'_1\ \tau'_1) \ldots (\texttt{tag}'_m\ \tau'_m))\ \rho_\uparrow) \\
\sqsubseteq_\epsilon & (\texttt{RECORDOF}\ ((\texttt{tag}_1\ \tau_{1\uparrow}) \ldots (\texttt{tag}_n\ \tau_{n\uparrow}))\ \rho_\downarrow)
\end{array}}
$$

$$
\frac{\tau_{\mathrm{body}\downarrow}\ \sqsubseteq_\epsilon\ \tau_{\mathrm{body}\uparrow}}{(\texttt{POLY}\ ((\texttt{dvar}\ \kappa) \ldots)\ \tau_{\mathrm{body}\downarrow})\ \sqsubseteq\ (\texttt{POLY}\ ((\texttt{dvar}\ \kappa) \ldots)\ \tau_{\mathrm{body}\uparrow})}
$$

$$
\frac{\mathcal{A}_{\sqsubseteq_\epsilon} \cup \{(\texttt{RECTYPE t}\ \tau_\downarrow) \sqsubseteq_\epsilon \tau_\uparrow\}\ \vdash_{\sqsubseteq_\epsilon}\ \tau_\downarrow[(\texttt{RECTYPE t}\ \tau_\downarrow)/\texttt{t}] \sqsubseteq_\epsilon \tau_\uparrow}{\mathcal{A}_{\sqsubseteq_\epsilon}\ \vdash_{\sqsubseteq_\epsilon}\ (\texttt{RECTYPE t}\ \tau_\downarrow) \sqsubseteq_\epsilon \tau_\uparrow}
$$

$$
\frac{\mathcal{A}_{\sqsubseteq_\epsilon} \cup \{\tau_\downarrow \sqsubseteq_\epsilon (\texttt{RECTYPE t}\ \tau_\uparrow)\}\ \vdash_{\sqsubseteq_\epsilon}\ \tau_\downarrow \sqsubseteq_\epsilon \tau_\uparrow[(\texttt{RECTYPE t}\ \tau_\uparrow)/\texttt{t}]}{\mathcal{A}_{\sqsubseteq_\epsilon}\ \vdash_{\sqsubseteq_\epsilon}\ \tau_\downarrow \sqsubseteq_\epsilon (\texttt{RECTYPE t}\ \tau_\uparrow)}
$$

The mysterious use of $\succeq$ (read "restricted to") is discussed in the next section.

The intuition here is that if the argument to an operator which has effect ϵ_ψ is expected to be immutable or if the procedure does not mutate any surface components of the mutable argument, then it is safe to pass as argument a value whose actual type is a subtype of that expected so long as the component types are likewise safe. If, on the other hand, the argument is expected to be surface mutable and the procedure expects to have the right to surface mutate it, then the component types must be strictly equivalent to the expected component types so they will denote values at least as high in the type inclusion ordering as any new value that we may update them to now contain.

The subtleties arise from ONEOF and RECORDOF types. In the case of a ONEOF, if equivalence is necessary then we can in fact allow the ONEOF to be mutated to vary over more fields (a supertype) so long as each field component is equivalent to the expected type. This is because any procedure which accesses the ONEOF must do so through a TAGCASE form, which requires an ELSE clause for cases where the tag of a ONEOF is not one of the anticipated tags. Normally, the ELSE clause would signal an error. For RECORDOF forms, we need the additional requirement that the number of fields match (*i.e.,* that the subtype have no extraneous fields).

6.3 Effects as Store Traversal Paths

So what is the meaning of the mysterious "restricted to" function used in $\epsilon_{\mathrm{mod}} \succeq \epsilon_{\mathrm{subr}\uparrow}$? Consider the SUBR ϵ-modulated inclusion axiom. If we are receiving a SUBR argument, we need to ϵ-modulate the inclusion of the body type against the expected body type. This is because if we invoke the procedure being passed in, we could gain access to mutable structure through its return value. We must be careful that if we expect to have mutation priveleges on the returned value that the mutation will be safe.

But what if the expected latent effect of the procedure parameter is not a sub-effect of the effect by which we are modulating? This would indicate that we are passing a procedure which will not be invoked, so there is no potential danger. In such a situation, $\epsilon_{\mathrm{mod}} \succeq \epsilon_{\mathrm{subr}\uparrow}$ is defined to be PURE; otherwise it is just ϵ_{mod}.

Similarly in the RECORDOF and ONEOF rules, if we are concerned with surface mutating a component of an aggregate structure but we notice that we do not in fact access any of its surface components (we do not READ any component), then again we can relax and perform unmodulated inclusion.

Now consider if the effects involved precisely reflect the effect collecting semantics of the procedures involved. That is, what if the effects were generated by the type and effect inference rules? Then there is more information in the effects than just sub-effect information. In fact, the structure of the effects will reveal the traversal path taken through the store by the procedures involved. For example, if the procedure we are passing has the latent effect ϵ whereas we were modulating with respect to an effect ϵ_{mod} for which ϵ is not a textual substring, then we could again deduce from the structure of the effects that the procedure passed is not in fact invoked. This is because the application type and effect rule and the λ type and effect rule are careful to preserve the structure of the latent effects of any procedures created or invoked. The effect masking rules would likewise preserve this structure.

In such cases where $\epsilon_{\mathrm{subr}\uparrow}$ is not a textual substring of ϵ_{mod}, again the $\succeq$ function computes PURE. However, where the substring does occur, $\succeq$ strips the prefix of ϵ_{mod} up to and including $\epsilon_{\mathrm{subr}\uparrow}$. This may involve gathering together the simultaneous restriction of several EITHER effects, which indicate a branching possibility with the procedure body. Any branch along which the $\epsilon_{\mathrm{subr}\uparrow}$ appears as a substring is still an active traversal path of interest. Notice that as traversal branches become inactive, they are truncated to PURE, at which point they can be dropped from an EITHER form. Note further that recursive procedures will necessitate denoting traversal path effects as possibly cyclic graphs (RECEFFECTS). Restricting a cyclic path would be tantamount to cycling through a DFA.

What is the utility of doing this traversal path restriction? First, whenever a traversal path is exhausted to PURE, the modulated inclusion test becomes a simple inclusion test. More useful though is that any prefix of a traversal path which is stripped by restriction will no longer influence the modulated inclusion test. Specifically, each ρ_{W} removed from the ϵ_{mod} relaxes the modulation.

In sum, we can follow the effect collecting semantics of a procedure in lockstep with the types of the parameters passed to it to verify the safety of potential side-effects to the arguments.

6.4 Summary of ϵ-Modulated Type Inclusion

In retrospect, we have chosen to maintain inclusion axioms for structured types based solely on an intuitive types-as-sets/subtypes-as-subsets characterization. We subsequently had to hone the application rule to tolerate argument subtyping modulated by the latent effect of the operator.

The FX-87 manual, by contrast, chose to develop structure type inclusion modulated solely on the mutability of the region of the structure components, without consideration of whether this is the type of an argument to an operator which actually has a WRITE effect on said region. Such a stance makes the inclusion axioms no longer be based on the intuitive notion of subtypes as subsets. More disturbing, it also permits the application soundness concern to bleed into the subtyping test even when an inclusion test does not involve argument/application inclusion of an operator which has a WRITE effect. This is unsettling since, for instance, two branches of an IF expression may fail to be in inclusion relation under the manual's axiomatization of inclusion whereas the more perceptive modulated formulation developed here would succeed. This has important implications for the set of programs which $*$-$\mathcal{TREK}$ will accept as well-types but which a strict manual-prescribed implementation would spuriously reject. Semantically rationalizing the set of programs rejected by a manual-prescribed implementation would therefore be, at best, haphazard.

7 Description Intersection for Sharing/Interference Detection

The issues of Effect Masking and ϵ-Modulated Inclusion in the face of polymorphic parameterization and projection were addressed in the preceding sections. We have not yet considered how effect annotations might be used to coordinate concurrent evaluation. Again the problem arises that projection can change the effect complexion of a polymorphic procedure's body by introducing heretofore unanticipated patterns of storage usage. Here our concern is that new patterns of interference and sharing may arise. In this section we formalize a Sharing/Interference Criterion upon which we can decide how to re-coordinate concurrency based on information revealed by projection.

We do so by first formalizing a test for whether two types describe values which may share or interfere, based on a formal notion of sharing regions and interfering effects. This test for sharing/interference I call *description intersection*, denoted $\cap$. The formal definition of $\cap$ is detailed in the following section, after which its incorporation into the Sharing/Interference Criterion is discussed.

7.1 Region-Based Sharing Detection

Region-based sharing detection is inductively defined over descriptions in terms of description intersection. The intuition is that two type expressions denote sharing/interfering data iff they intersect. The intersection rules, and the intuitions behind them, are presented in the following subsections.

7.1.1 Basic Intersection Axioms

The following axioms define $\cap$ to be transitive and symmetric and to respect $\equiv$:

$$\begin{array}{rcl} \delta \cap \delta' \;\wedge\; \delta' \cap \delta'' & \Longrightarrow & \delta \cap \delta'' \\ \delta \cap \delta' & \Longrightarrow & \delta' \cap \delta \\ \delta \equiv \delta' \;\wedge\; \delta' \cap \delta'' & \Longrightarrow & \delta'' \cap \delta''\{\delta'/\delta\} \end{array}$$

...where $\delta''\{\delta'/\delta\}$ denotes δ'' with the substitution of δ' for any or all occurrences of δ within δ''.

Notice that $\cap$ is not a congruence relation since it is *not* reflexive. Specifically, as we shall see: @= $\not\cap$@=, PURE $\not\cap$PURE, (ALLOC ρ) $\not\cap$(ALLOC ρ), (READ ρ) $\not\cap$(READ ρ), and $\tau_{\mathrm{base}} \not\cap \tau_{\mathrm{base}}$.

7.1.2 Region Intersection

Insofar as region expressions denote sets of regions [Lucassen 87, p.35], region expression intersection is trivially defined: it is set intersection over the denoted regions (where region literals are interpreted as unit sets containing only that literal);

however, we will be interested only in the cases where two region expressions intersect at some region inequivalent to the immutable region (`@=`). I use $\cap$ to implicitly denote such cases. Should it become necessary to consider region intersection where the point of intersection can be the immutable region [it won't], I would use $\cap^{+\texttt{@=}}$. Formally,

$$\texttt{(RUNION}\ \rho\ \rho'\texttt{...)} \cap \texttt{(RUNION}\ \rho\ \rho''\texttt{...)} \qquad \text{for}\ \rho \not\equiv \texttt{@=}$$

7.1.3 Effect Intersection

Two effect expressions intersect iff they denote interfering effects. Specifically, `WRITE`s to a region intersect with `READ`s or `WRITE`s to intersecting regions, while two `MAXEFF`s intersect iff any component effect of one intersects with any component effect of the other. Note that the `PURE` effect does not intersect with any effect, including itself; similarly for `ALLOC` effects. Formally,

$$\texttt{(MAXEFF (WRITE}\ \rho\texttt{)}\ \epsilon\texttt{)} \quad \cap \quad \texttt{(MAXEFF (READ}\ \rho\texttt{)}\ \epsilon'\texttt{)}$$

$$\texttt{(MAXEFF (WRITE}\ \rho\texttt{)}\ \epsilon\texttt{)} \quad \cap \quad \texttt{(MAXEFF (WRITE}\ \rho\texttt{)}\ \epsilon''\texttt{)}$$

7.1.4 Type Intersection

Type expressions comprise a highly structured and semantically rich domain. Not surprising, therefore, type expression intersection is more complicated than region or effect intersection. The intuition, however, is that two type expressions intersect iff the collection of values which one of them approximates can share mutable storage with the collection of values which the other approximates.[14]

In the following, we will have occasion to consider, given two type expressions, whether one is a subtype ($\sqsubseteq$) of the other, without deference to which is the subtype and which the supertype. In such situations, we are said to be asking whether the two type expressions are *comparable* ($\underline{\square}$). If two type expressions are not comparable, they are said to be *incomparable* ($\not\underline{\square}$).

I present the type expression intersection rules via structural induction over the type domain.

$$\frac{\rho \cap \rho' \ ,\ \tau \ \underline{\square}\ \tau'}{\texttt{(ONEOF ((tag}\ \tau\texttt{)...)}\ \ \rho\texttt{)} \cap \texttt{(ONEOF ((tag}\ \tau'\texttt{)...)}\ \ \rho'\texttt{)}}$$

$$\frac{\tau \cap \tau'}{\texttt{(ONEOF ((tag}\ \tau\texttt{)...)}\ \ \rho\texttt{)} \cap \tau'}$$

$$\frac{\rho \cap \rho' \ ,\ \forall i,j\ .\ \tau_i \ \underline{\square}\ \tau_j}{\begin{array}{ll} & \texttt{(RECORDOF ((tag}_1\ \tau_1\texttt{)...(tag}_n\ \tau_n\texttt{))}\ \rho\texttt{)} \\ \cap & \texttt{(RECORDOF ((tag}_1\ \tau_1'\texttt{)...(tag}_n\ \tau_n'\texttt{))}\ \rho'\texttt{)} \end{array}}$$

[14] This is simply the question of whether the *greatest lower bound* (glb) of two type expressions encompasses common mutable storage. This glb is guaranteed to exist for any two type expressions since the type domain is a pointed partial ordering with bottom ($\bot$) element VOID. By contrast, the *least upper bound* (lub) of two type expressions does *not* in general exist; for example, INT and BOOLE have no lub in our type domain.

$$\frac{\tau \cap \tau'}{\texttt{(RECORDOF ((tag}\ \tau\texttt{)...)}\ \ \rho\texttt{)} \cap \tau'}$$

Note that although an instance of a `RECORDOF` may possess more fields than specified by the type under which it is accessed, those extraneous fields will not be accessible under this type. Thus, although the type expressions associated with these extraneous fields may indeed intersect with other type expressions, this intersection will not be observable. It is therefore not considered in the intersection inference rules.

$$\frac{\tau_{\mathrm{rtn}} \cap \tau \quad \vee \quad \rho_{\mathrm{R}}(\epsilon) \cap \rho_{\mathrm{W}}(\tau) \quad \vee \quad \rho_{\mathrm{W}}(\epsilon) \cap (\rho_{\mathrm{R}}(\tau) \cup \rho_{\mathrm{W}}(\tau))}{\texttt{(SUBR}\ \epsilon\ \ (\tau_{\alpha 1}\ \ \ldots)\ \ \tau_{\mathrm{rtn}}\texttt{)} \cap \tau}$$

The rule for `SUBR` forms checks those mutable storage locations over which the procedure may be closed (since these locations may be accessed or mutated by invocations to the procedure) to see if any manipulation of the one may generate interference with respect to manipulation of the other, for example if invoking the procedure may cause mutation of some common shared storage in τ. Notice that we can ignore the $\tau_{\alpha i}$ since if their ρ's do not appear in the latent effect ϵ or in τ_{rtn} then they are not effectively used; however we cannot ignore ϵ or τ_{rtn} themselves. Specifically, we must consider (i) the `READ`/`WRITE` regions in ϵ (because they may indicate regions of access or mutation to local mutable storage over which the procedure is closed which may share with τ), and (ii) τ_{rtn} (since those components of τ_{rtn} which do not intersect with ϵ may nonetheless indicate external storage over which the `SUBR` was closed but whose components the `SUBR` does not provide procedural `READ` or `WRITE` access to, which is to say that the `SUBR` merely returns an anonymous alias for the external mutable storage so that invocations of the procedure reveal this storage). Of course, a corollary to this is that a `SUBR` which commits no `READ` or `WRITE` effects and which returns no mutable storage cannot intersect anything, including itself.

$$\frac{\mathcal{A}_{\emptyset} \cup \{\texttt{(RECTYPE t}\ \ \tau_{\downarrow}\texttt{)}\ \not\cap \tau_{\uparrow}\} \quad \vdash_{\cap} \quad \tau_{\downarrow}[\texttt{(RECTYPE t}\ \ \tau_{\downarrow}\texttt{)/t}] \cap \tau_{\uparrow}}{\mathcal{A}_{\emptyset} \quad \vdash_{\cap} \quad \texttt{(RECTYPE t}\ \ \tau_{\downarrow}\texttt{)} \cap \tau_{\uparrow}}$$

$$\frac{\mathcal{A}_{\emptyset} \cup \{\tau_{\downarrow}\ \not\cap \texttt{(RECTYPE t}\ \ \tau_{\uparrow}\texttt{)}\} \quad \vdash_{\cap} \quad \tau_{\downarrow} \cap \tau_{\uparrow}[\texttt{(RECTYPE t}\ \ \tau_{\uparrow}\texttt{)/t}]}{\mathcal{A}_{\emptyset} \quad \vdash_{\cap} \quad \tau_{\downarrow} \cap \texttt{(RECTYPE t}\ \ \tau_{\uparrow}\texttt{)}}$$

The `RECTYPE` axiom is a sequent rule which maintains an assumption list as for the `RECTYPE` inclusion axiom [§ 3.3.7]. Here, the assumption is that no intersection occurs. Any contradiction terminates the test with a positive intersection.

$$\frac{\tau \cap \tau'}{\texttt{(POLY ((param}\ \kappa\texttt{)...)}\ \ \tau\texttt{)} \cap \tau'}$$

The `POLY` rule, by contrast, is straightforward. The only subtlety is that the `param` can be assumed not to intersect anything, including itself, by Soundness of the Region Anti-Aliasing Rule [See `Dvars` below]).

7.1.5 POLY Params: Abstract Description Intersection

`POLY`/Pλ `params` cannot intersect anything since nothing can be done through them which would constitute access or mutation to mutable storage. The Region Anti-Aliasing Rule ensures the soundness of this assumption by guarding against projections that would result in intersections producing heretofore unanticipated interference. In the case of such an unsavory projection, appropriate action would be taken to ensure proper serialization of processes with respect to these newly sharing/interfering data.

7.1.6 Soundness Claim

Only those type expressions which intersect according to the above inference rules and informal considerations can possibly share mutable storage. The proof is by soundness of the above axioms and by structural induction on type expressions.

7.1.7 A Plea for Well-Disciplined Region Coloring

Consider the run-time complexity of the above induced type intersection algorithm. Specifically, consider the complexity of testing for intersection of two `RECORDOF` types of length n.

The number of subexpressions in a `RECORDOF` type of length n is $O(n)$. For each subexpression of the type, we must check intersection against each subexpression of the other `RECORDOF` type expression. The cost of each of these $O(n^2)$ subtasks is the cost of a region intersection test (call it $O(n)$ for a bit-vector intersection test) plus a $\sqsubseteq$ test (call it $O(n^2)$ since recursive types may require $\mathrm{lcm}(n, m)$ to check two types of lengths n and m). Disregarding the possibility of memoization, this yields an overall running time of $\boxed{O(n^4)}$.

Now consider if the programmer used well-disciplined region coloring to color the store so as to have overlapping regions only in potentially overlapping sets of values. Now intersection can be tested merely by testing the intersection of the sets of regions appearing in the two types. By contrast to the type intersection algorithm, merely checking intersection of two sets of regions of size $O(n)$ can be done in time $\boxed{O(n)}$ using a bit-vector hashing intersection test.

This represents a dramatic improvement on the type intersection algorithm for large multi-colored descriptions. It also provides precise and verifiable documentation of the designer's intended permissible sharing among mutable structures, a no less desirable side-benefit.

Unfortunately there is a trade-off between the number of regions introduced by the programmer to admit concurrency and the running time of sharing/interference detection. For concurrency and precise effect masking, the user should maximally distribute data structures over regions so the region information carries exactly the sharing information. Specifically, the user should not restrict the number of regions used to the number of processors on the machine (the naïve user may be tempted to do this) since sharing information can be more important than processor distribution.

7.2 The Region Anti-Aliasing Rule

When compiling a procedure body, a compiler uses the body's effect information to coordinate concurrent evaluation of non-interfering expressions. Unfortunately, within the body of a polymorphic procedure, not all type/effect/region information is available to the compiler since this information can be polymorphized over. Indeed, this is the very reason for providing polymorphism in the language at all: to share common code across distinct type/effect/regions.

When faced with the compilation of a polymorphic procedure, then, the compiler writer has many choices for how to schedule the concurrency with respect to this abstracted information. First, one could assume that all abstracted information will potentially interfere with all other expressions within the procedure body, and hence serialize all expressions which mention the parameters. Call this the *minimal parallelism* approach.

Alternatively, one could assume that no parameters will share/interfere with anything in the procedure body. Of course, this requires enforcement of the assumption at each projection (`PROJ`). Call this the *maximal parallelism* approach.

These two approaches delimit the two extremes of a continuum of possibilities. Each distinct interference pattern contained within these boundaries is a feasible interference pattern and could be precisely implemented with distinct concurrent scheduling. Unfortunately, the number of such patterns for n parameters is proportional to the size of the powerset of an n-element set, which is $O(2^n)$. We certainly would not want to compile $O(2^n)$ versions of each polymorphic procedure.[15]

The FX-87 Manual prescribes the maximal parallelism approach. The rule which enforces the no-new-interference assumption is called the Region Anti-Aliasing Rule. It reads:

> There is an important restriction on the way a polymorphic expression may be projected onto different arguments of kind `REGION` and `EFFECT`: all the mutable region arguments must be mutually disjoint and must not intersect with free mutable region variables or constants used inside the types of the body of the polymorphic expression. [Gifford *et al.* 87, p.40]

Observe that there is no restriction on parameters of kind `TYPE`. This is because there is no effective way to manipulate a datum of polymorphic type within the `POLY` body. Rather, one would accept a procedure argument for accessing it, polymorphizing the procedure over the effect that it would incur when invoked.

This rule yields a crude, very conservative approximation to detecting heretofore unanticipated interference. In light of the discussion of interference in the preceding section, our first attempt at improving this rule would be to sharpen it to consider only the ρ_{R} and ρ_{W} [§ 4.2] rather than all region literals and constants. This at least does not cause false conservatism with respect either to argument types of `SUBR`s or to `ALLOC` regions.

There is a more substantial problem with FX-87's treatment of this issue though: it relegates to the domain of illegal code all projections that would violate this rule. This is an unacceptable stance, and one which the language designers themselves have subsequently relaxed. Instead, the Region Anti-Aliasing Rule has been demoted to a choice metric for using one of two compiled versions of a polymorphic procedure: either the maximally parallel one or the minimally parallel one. Upon detection of potential new interference, the minimally parallel code is used, otherwise the maximally parallel code.

This, however, is not sufficient for the serious parallel programmer. A more precise new-interference detector is

[15] Although the language Self takes an interesting stance of run-time compiling (and later garbage collecting) customized code as needed [Chambers, Ungar & Lee 89].

needed. For instance, one might like better detection of sharing/interference than mere region intersection provides, for instance employing description intersection as defined earlier. This could be used to detect if the particular projection choices of effect and region, when instantiated into the polymorphic type of the expression body, actually give rise to *new* patterns of interference.

Note carefully the emphasis on the qualifier *new*. Specifically, if we choose to project some abstracted region onto say `@RED`, but that region parameter is used only within `RUNION` expressions involving say `@BLUE`, then if instantiating the parameter reveals interference or sharing with respect to some other datum allocated in a region encompassing both `@RED` and `@BLUE`, then this instantiation would not give rise to any interference patterns which the maximally parallelized code has not already had to account for. A similar argument holds for instantiation of effects which are elements of `MAXEFF` forms.

Of course, implementing an anti-aliasing rule to this level of precision could prove expensive since, as noted earlier, general description intersection can be expensive. However, the concern for precise detection of only *new* patterns of sharing/interference is not prohibitively expensive. To wit, when a polymorphic procedure is compiled, the analysis can be performed which associates each poly parameter with those regions which it is already consistently associated with. The anti-aliasing rule can then filter out projection arguments to these associates before testing for intersection with other regions and effects.

For a still more precise treatment, the compiler could associate with each region/effect parameter only those choices of projection argument which *would* result in new interference patterns based on a general description intersection analysis of the specific occurrences of the parameters within the body type. Note that this would in general include restrictions on intersections among the projection arguments themselves.

Clearly, there are many choices for precision/expense trade-offs. Probably a lazy, gradually refining region anti-aliasing rule would be in order. Such a strategy would, for example, always try the crude easy anti-aliasing rule prescribed by the manual first. If it detects potential new interference, then gradually more expensive tests for interference could be attempted until either it is realized that no new interference is in fact taking place or that no amount of refining of the test will reverse the decision that it *is* taking place. Moreover, since the failure of anti-aliasing need not result in rejected code (merely possibly missed opportunities for concurrency), compiler directives (pragmas) could be provided to suggest how aggressive an attempt should be made on any particular projection, or even on any particular polymorphic procedure since expressions which are not believed by the programmer to be a major parallelism opportunity or bottleneck should not slow the overall compilation time for trivial pursuits.

8 Summary

To conclude, we have investigated four views of effect annotations: raw, pickled, traversal-path, and sharing/intersection. Raw effects are based on a storage cell view of effects and is useful for precise effect masking. Pickled effects take a process synchronization view of effects, which is useful for permitting flexible subtyping of procedural arguments in applications. The traversal-path view of effects, though less developed here, annotates a collecting semantics of the storage cells manipulated in the course of evaluating an expression. It is useful for ensuring the safety of passing aggregate data to mutator procedures. Finally, we investigated the sharing/interference information encoded in effect annotations. It was useful for coordinating concurrency in evaluating the bodies of polymorphic procedures. Throughout, we have embraced the subtypes-as-subsets view of type inclusion, dealing with safety and concurrency concerns separately by taking various views of effect annotations.

9 Related Work

This work was based heavily on the languages FX-87 [Gifford *et al.* 87] and MFX [Lucassen 87], as has been detailed throughout the paper. I are not aware of other languages which employ a polymorphic region-based effect system, but our stratification into types and kinds grows out of the language LS1 [McCracken 84]. This style has also been used in AMBER [Cardelli 84/85/86] and QUEST [Cardelli 89].

Unlike traditional approaches to side-effect analysis and sharing analysis, our region-based effect system does not give `READ`/`WRITE` annotation at the level of precision of specific locations or even of specific variable instances. Rather, it propagates information based on the region information embedded within each variable's type and binding region. This will not be as precise as traditional static analysis of aliasing and sharing information [Banning 78/79] [Barth 77/78] [Rosen 79], but such traditional approaches have been shown to be P-space hard [Weihl 80] or even NP-complete or co-NP-complete [Myers 81]. Using region and effect annotations seems a reasonable trade-off in light of the potential expense of these more aggressive methods. Moreover, using region and effect annotations significantly reduces the sheer engineering effort otherwise necessary to implement a traditional approach [Cooper, Kennedy & Torczon 86].

Finally, one might wish to perform more detailed sharing analysis than a region-based analysis can provide. For instance, one might wish to accumulate `SHARES` information about specific local variables to provide sharing information at the level of specific values rather than over entire regions [Sturgis 87]. Unfortunately, this level of analysis must be sensitive to the specific flow of control during program execution to detect information like `READ` after `WRITE` and the like [Aho & Ullman 86, Chapt.10]. Unfortunately, in a language which supports procedure parameters with potential side-effects, this sort of analysis becomes excruciatingly difficult. Shivers has summarized the problem nicely [Shivers 88, p.165]:

> So with regard to flow analysis in LISP, we are faced with the following unfortunate situation:
>
> - In order to do flow analysis, we need a control flow graph.
> - In order to determine control flow graphs, we need to do flow analysis.
>
> Oops.

Perhaps region-based effect systems offer a palatable alternative. A similar approach, based on *liabilities* rather than on regions is currently under development at Yale University [Guzmán & Hudak 90].

Acknowledgments

I am indebted to John Pezaris for helping me to organize these ideas. I have also benefited from many useful discussions with Pierre Jouvelot, Mark Sheldon, and James O'Toole. Also, Jonathan Rees and Franklyn Turbak encouraged me to develop these results.

Most importantly, Prof. David A. McAllester was instrumental in helping me solidify these ideas. Specifically, it was he who first suggested to me that description inclusion be defined solely on an intuitive subtypes-as-subsets analogy, leaving the issue of application soundness as a separate issue.

References

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

[Banning 78/79]
John P. Banning
An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables
In Proc. ACM POPL 1979, pp.29–41, as condensed from
A Method for Determining the Side Effects of Procedure Calls
Ph.D. Thesis, Stanford University Report No.213, Stanford Linear Accelerator Center, Aug 78.

[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

[Barth 77/78]
Jeffrey M. Barth
A Practical Interprocedural Data Flow Analysis Algorithm
In CACM Vol.21 No.9, Sept. 1978, pp.724–736, as condensed from
A Practical Interprocedural Data Flow Analysis Algorithm and its Application
Ph.D. Thesis, CS TR No.770520, UC Berkeley, May 1977

[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.

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

[Chambers, Ungar & Lee 89]
Craig Chambers, David Ungar, and Elgin Lee
An Efficient Implementation of SELF*, a Dynamically-Typed Object-Oriented Language Based on Prototypes*
In Proc. ACM OOPSLA '89, Oct 1989, pp.49–70

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

[Cooper, Kennedy & Torczon 86]
Keith D. Cooper, Ken Kennedy, and Linda Torczon
The Impact of Interprocedural Analysis and Optimization in the $\mathbf{R}^n$ *Programming Environment*
In ACM TransPLaS, Vol.8 No.4, Oct. 1986, pp.491–523

[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

[Guzmán & Hudak 90]
Juan C. Guzmán and Paul Hudak
Single-Threaded Polymorphic Lambda Calculus
In Proc. IEEE LICS 1990, pp.333–343

[Harrison 89]
Williams Ludwell Harrison III
The Interprocedural Analysis and Automatic Parallelization of SCHEME *Programs*
In LISP and Symbolic Computation: An International Journal, Vol.2, No.3/4, Oct 1989, pp.179–396

[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 79]
Nancy McCracken
An Investigation of a Programming Language with a Polymorphic Type Structure
Ph.D. Thesis, Syracuse University, Jun 1979

[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

[Myers 81]
Eugene W. Myers
A Precise Inter-Procedural Data Flow Algorithm
In Proc. ACM POPL 1981, pp.219–230

[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.

[Rosen 79]
Barry K. Rosen
Data Flow Analysis for Procedural Languages
In JACM Vol.26 No.2, Apr 1979, pp.322–344

[Shivers 88]
Olin Shivers
Control Flow Analysis in SCHEME
In Proc. ACM SIGPLAN PLDI 1988, pp.164–174

[Stoy 77]
Joseph E. Stoy
Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory
(The MIT Press Series in Computer Science)
The MIT Press, 1977.

[Sturgis 87]
Howard E. Sturgis
Maintaining the Illusion of a Functional Language in the Presence of Side Effects
Xerox PARC TR CSL-88-2, Jul 1988

[Weihl 80]
William E. Weihl
Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables, and Label Variables (Extended Abstract)
In Proc. ACM POPL 1980, pp.83–94, as condensed from
MIT/LCS/TR-247, Jan. 1980