Electronic Notes in Theoretical Computer Science 70 No. 1 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume70.html 25 pages

Implementing Compositional Analysis Using Intersection Types With Expansion Variables

Assaf Kfoury [1]

Boston University

Geoffrey Washburn [2]

Boston University

J. B. Wells [3]

Heriot-Watt University

Abstract

A program analysis is *compositional* when the analysis result for a particular program fragment is obtained solely from the results for its immediate subfragments via some composition operator. This means the subfragments can be analyzed independently in any order. Many commonly used program analysis techniques (in particular, most abstract interpretations and most uses of the Hindley/Milner type system) are not compositional and require the entire text of a program for sound and complete analysis.

System $\mathbb{I}$ is a recent type system for the pure λ-calculus with intersection types and the new technology of expansion variables. System $\mathbb{I}$ supports compositional analysis because it has the *principal typings* property and an algorithm based on the new technology of β-unification has been developed that finds these principal typings. In addition, for each natural number k, typability in the rank-k restriction of System $\mathbb{I}$ is decidable, so a complete and terminating analysis algorithm exists for the rank-k restriction.

This paper presents new understanding that has been gained from working with multiple implementations of System $\mathbb{I}$ and β-unification-based analysis algorithms. The previous literature on System $\mathbb{I}$ presented the type system in a way that helped in proving its more important theoretical properties, but was not as easy for implementers to follow as it could be. This paper provides a presentation of many aspects of System $\mathbb{I}$ that should be clearer as well as a discussion of important implementation issues.

1 Introduction

Program analysis is useful for many different purposes, e.g., verifying that a program adheres to a specification, detecting error conditions statically, or generating information to be used by a compiler in optimization. Although the benefits of modularity in software engineering are well known, many commonly used program analysis techniques (in particular, most abstract interpretations [5] and most uses of the Hindley/Milner type system [14]) require the complete text of a program for sound and complete analysis. This is at odds with the desire (and, increasingly, the *need*) to design, implement, and assemble software in a modular, bottom-up manner. More and more often, large software systems are assembled from components that are designed separately and updated at different times. As most of the common program analysis techniques are not linear in time or space complexity, requiring the reanalysis of an entire program due to a single line change can become very costly as project size increases.

Ideally, program analysis would be done in a *compositional* way, where the analysis result for a particular program fragment is obtained solely from the results for its immediate subfragments via some composition (i.e., combining) operator. This means the subfragments can be analyzed independently of each other and in any order. When a system changes, unchanged fragments need not be reanalyzed. If a system is viewed as a tree where each internal node is the use of a composition operator, then only the changed subtree and each of its ancestor nodes would need to be reanalyzed, and in this case the program analysis is also *incremental*. The advantage of this kind of analysis is that local changes in the program require minimal global reanalysis. A fully compositional analysis is also much more easy to carry out in a *parallel* and *distributed* manner.

System $\mathbb{I}$ is a recent type system for the pure λ-calculus with intersection types and the new technology of expansion variables [11]. System $\mathbb{I}$ supports compositional analysis because it has the *principal typings* property and an algorithm based on the new technology of β-unification has been developed that finds these principal typings. (It is important not to confuse principal typings [17,8] with the much weaker property of the Hindley/Milner type system often referred to (erroneously) as "principal types".) Thus, if a term can be assigned a typing in System $\mathbb{I}$, then it can be assigned a *principal* typing and in the case of System $\mathbb{I}$ this means that every other possible typing for that term can be obtained via substitution. Therefore, once a principal

[1] Partly supported by NATO grant CRG 971607, NSF grant CCR 9988529, and Sun Microsystems equipment grant EDUD-7826-990410-US.
[2] Partly supported by NATO grant CRG 971607, NSF grant ITR 0113193, and Sun Microsystems equipment grant EDUD-7826-990410-US.
[3] Partly supported by EC FP5 grant IST-2001-33477, EPSRC grants GR/R 41545/01 and GR/L 36963, NATO grant CRG 971607, NSF grant CCR 9988529, and Sun Microsystems equipment grant EDUD-7826-990410-US.

typing has been inferred for a term, it is not necessary to ever analyze that particular term again. An important expected future benefit (work still to be done) of System-$\mathbb{I}$-style type inference is the real-time incremental analysis of programs as they are edited and changed.

Unfortunately the existing literature [11,10,9] on System $\mathbb{I}$ presents the type system in a way that helps in proving its more important theoretical properties, but is not as easy for implementers to follow as it could be. Because the algorithms behind System $\mathbb{I}$ have now been implemented several times [16], we can now better explain System $\mathbb{I}$ given the insights obtain from developing and using these implementations. In addition, we also provide advice in how one should proceed in implementing System $\mathbb{I}$.

2 Understanding Type Inference in System $\mathbb{I}$

2.1 Bare Minimum of System $\mathbb{I}$ Definitions for Examples

This subsection presents the bare minimum of the definitions of System $\mathbb{I}$ necessary to follow the following examples. The definition of System $\mathbb{I}$ starts from 3 syntactic categories. First, the language is the terms of the pure λ-calculus, denoted by Term and specified by the following pseudo-grammar:

$$M, N \in \mathsf{Term} \ ::= \ x \mid \lambda x.M \mid MN$$

where x is a *variable*, $\lambda x.M$ is an *abstraction*, and MN is an *application*.

Second, the types are from the set Type specified by the pseudo-grammar:

$$\begin{aligned} \bar{\tau} \in \mathsf{Type}^{\rightarrow} \quad &::= \quad \alpha \mid \tau \rightarrow \bar{\tau} \\ \tau \in \mathsf{Type} \quad &::= \quad \bar{\tau} \mid \tau_1 \wedge \tau_2 \mid F\tau \end{aligned}$$

where α is a *type variable*, $\tau \rightarrow \bar{\tau}$ is a *function type*, $\tau_1 \wedge \tau_2$ is an *intersection type*, and $F\tau$ is the application of an *expansion variable* F to a type τ. Types involve two kinds of variables: type variables and expansion variables. Types are stratified into two levels, $\mathsf{Type}^{\rightarrow}$ and Type, in order to force uses of the intersection type constructor and expansion variable applications to only appear in the domain of function types. An intersection type $\tau_1 \wedge \tau_2$ abstractly indicates that a value of that type is used in two different contexts within a term, one requiring type τ_1 and the other type τ_2. Expansion variables provide a means to delay "expanding" the type of a term into an intersection type until more is known about whether it will be used in more than one context.

The third syntactic category of System $\mathbb{I}$ is the set of expansions $\mathsf{Expansion}$, which is specified by the pseudo-grammar

$$e \in \mathsf{Expansion} \ ::= \ \Box \mid e_1 \wedge e_2 \mid Fe$$

where the symbol $\Box$ stands for a hole into which a type can be inserted. The expression $e[\tau_1, \ldots, \tau_n]$ denotes the result of filling the $n \geq 1$ holes of the expansion e with n types $\tau_1, \ldots, \tau_n$, from left to right respectively. When an expansion e with $n \geq 1$ holes is substituted for the expansion variable F in the type $F\,\tau$, we insert n copies of τ into the n holes of e, where each copy of τ has all of its type and expansion variables renamed fresh. Discussion of the precise details of how this variable renaming is carried out is postponed until section 3.

2.2 Examples of Inference

A good way of understanding how a complex system such as System $\mathbb{I}$ works is to see it in operation. In the following text we consider type inference for the very simple term $((\lambda x.xx)y)$, and the different approaches one may take within the framework provided by System $\mathbb{I}$.

Although quite simple, the term $((\lambda x.xx)y)$ has two features that illustrate important differences with type-inference in the style of the algorithm $\mathcal{W}$ [14] (or one of its variants) for the Hindley/Milner type system. First, $((\lambda x.xx)y)$ is an open term, i.e., it has a free variable. Second, algorithm $\mathcal{W}$ can not infer a typing for $((\lambda x.xx)y)$. Although algorithm $\mathcal{W}$ can infer a typing for the observationally equivalent term ($\mathsf{let}\ x = y\ \mathsf{in}\ xx$), the resulting analysis is not compositional — algorithm $\mathcal{W}$ must analyze the definition (here it is y) of the let-bound variable x before the body (xx) can be analyzed. System $\mathbb{I}$ has no such limitation, as shown below using this example.

2.2.1 Bottom-Up Constraint Collection

One approach to inference in System $\mathbb{I}$ consists in recursively processing the term from the leaves at the bottom (i.e., variable occurrences) to the root at the top (the full term), collecting constraints between types along the way, and then solving the constraints afterward. This approach is sufficient for some purposes and simple to define, but results in a non-compositional algorithm.

Below we step through the process of constructing a typing derivation tree for our chosen term. Rather than immediately building a typing derivation, we build instead an *analysis tree*, which represents a potential typing derivation, provided the associated typing constraints can be solved. Because the analysis tree is built from the leaves up to the root, in intermediate steps we are actually operating on an *analysis forest*, i.e., a collection of analysis trees.

Each node in an example analysis tree is a pair $n :: r$ of a *typing rule name* n and an *analysis result* r. An analysis result r is in turn a pair t/Δ of a *typing* t and a *typing constraint set* Δ. The intended meaning is that a solution for the constraint set will also make the typing valid for the λ-term being analyzed. A typing t is a pair $\langle A, \tau\rangle$ of a *type environment* A (formally defined later) and a result type τ. A typing constraint set Δ is a set of typing constraints, each constraint being of the form $\tau \doteq \tau'$. A constraint of the form $\tau \doteq \tau$ with both sides equal is *solved.* The examples below follow the convention that solved constraints are not shown. Furthermore, constraint sets containing only solved constraints are sometimes omitted completely together with the preceding "/".

The typing rules used are such that in the examples below, every leaf node is labeled with a typed term variable $x^{\bar{\tau}}$, every application node is labeled with $@^{\bar{\tau}}$, and every λ-abstraction node (corresponding to the λ-binding of a variable x) with λx (or λx^{τ} if the bound variable does not occur in the function body). In addition, accounting for the possibility that an argument may be used at different types (not yet determined) in the body of a function, every subterm occurrence in argument position gives rise to a node labeled with a fresh expansion variable F.

The process starts by building the analysis forest from the leaves of the term (new nodes being added to the analysis forest are indicated by enclosing them in a solid box):

$$\boxed{x^{\alpha_1} :: \langle \{x \mapsto \alpha_1\}, \alpha_1 \rangle \,/ \varnothing} \quad \boxed{x^{\alpha_2} :: \langle \{x \mapsto \alpha_2\}, \alpha_2 \rangle \,/ \varnothing} \quad \boxed{y^{\alpha_3} :: \langle \{y \mapsto \alpha_3\}, \alpha_3 \rangle \,/ \varnothing}$$

The environment in the typing for an occurrence of variable x contains a single mapping from x to a fresh type variable α_i, which is also the type derived for this occurrence of x. There is a different typing for every occurrence of the same variable x. No constraint is generated by the typing for a variable occurrence.

The next node we add to the analysis forest is an expansion variable:

$$x^{\alpha_1} :: \langle \{x \mapsto \alpha_1\}, \alpha_1 \rangle \,/ \varnothing \quad \begin{array}{c} \boxed{F_1 :: \langle \{x \mapsto F_1\alpha_2\}, F_1\alpha_2 \rangle \,/ \varnothing} \\ | \\ x^{\alpha_2} :: \langle \{x \mapsto \alpha_2\}, \alpha_2 \rangle \,/ \varnothing \end{array} \quad y^{\alpha_3} :: \langle \{y \mapsto \alpha_3\}, \alpha_3 \rangle \,/ \varnothing$$

In preparation for a term to be an argument of an application, we wrap that term with an expansion variable F_i; substituting an expansion e for F_i later allows the argument to be used in multiple contexts in the body of the function that consumes it. Wrapping the argument with an expansion variable, we are able to analyze the argument independently of the function. Expansion variables are important for implementing the compositionality of the analysis. As bindings in the environment of the argument may be used by the consuming function, all types in the argument environment are also wrapped with the expansion variable.

The next node is an application node:

$$
\begin{array}{ccc}
 & \boxed{@^{\beta_1} :: \langle\{x \mapsto \alpha_1 \wedge F_1\alpha_2\}, \beta_1\rangle \,/\, \{\alpha_1 \doteq F_1\alpha_2 \to \beta_1\}} & \qquad y^{\alpha_3} :: \langle\{y \mapsto \alpha_3\}, \alpha_3\rangle \,/\, \varnothing \\
x^{\alpha_1} :: \langle\{x \mapsto \alpha_1\}, \alpha_1\rangle \,/\, \varnothing & F_1 :: \langle\{x \mapsto F_1\alpha_2\}, F_1\alpha_2\rangle \,/\, \varnothing & \\
 & x^{\alpha_2} :: \langle\{x \mapsto \alpha_2\}, \alpha_2\rangle \,/\, \varnothing &
\end{array}
$$

As an application node has two children, the same variable x may have a type binding in the environments of both children. As a result, when the two environments are merged, the new environment assigns to x the intersection of its types in the two branches. Every application node introduces a constraint, written $\tau_1 \doteq \tau_2 \to \beta$, indicating that the type τ_1 of the function branch must be a function type, whose domain must be made equal to the result type τ_2 of the argument and whose range must be made equal to the fresh type variable β.

Next, there are two new nodes, one for the λ-abstraction $(\lambda x.xx)$ and one corresponding to wrapping the typing of y with a fresh expansion variable F_2:

$$
\begin{array}{cc}
\boxed{\lambda x :: \langle\varnothing, \alpha_1 \wedge F_1\alpha_2 \to \beta_1\rangle \,/\, \{\alpha_1 \doteq F_1\alpha_2 \to \beta_1\}} & \boxed{F_2 :: \langle\{y \mapsto F_2\alpha_3\}, F_2\alpha_3\rangle \,/\, \varnothing} \\
@^{\beta_1} :: \langle\{x \mapsto \alpha_1 \wedge F_1\alpha_2\}, \beta_1\rangle \,/\, \{\alpha_1 \doteq F_1\alpha_2 \to \beta_1\} & y^{\alpha_3} :: \langle\{y \mapsto \alpha_3\}, \alpha_3\rangle \,/\, \varnothing \\
x^{\alpha_1} :: \langle\{x \mapsto \alpha_1\}, \alpha_1\rangle \,/\, \varnothing \qquad F_1 :: \langle\{x \mapsto F_1\alpha_2\}, F_1\alpha_2\rangle \,/\, \varnothing & \\
x^{\alpha_2} :: \langle\{x \mapsto \alpha_2\}, \alpha_2\rangle \,/\, \varnothing &
\end{array}
$$

The type inferred for a λ-abstraction $\lambda x.M$ is the function type $\tau_1 \to \tau_2$ whose domain is the type τ_1 of x in the environment (before it is discharged) and whose range is the result type τ_2 inferred for M. If in a λ-abstraction $\lambda z.M$ there are no free occurrences of z in M (not in this example), the inferred type for $\lambda z.M$ is $\alpha_i \to \tau_2$ for some fresh type variable α_i, and the environment remains unchanged.

The last node is an application node, which introduces a new constraint, as shown:

$$
\begin{array}{c}
\boxed{@^{\beta_2} :: \langle\{y \mapsto F_2\alpha_3\}, \beta_2\rangle \,/\, \{\alpha_1 \wedge F_1\alpha_2 \to \beta_1 \doteq F_2\alpha_3 \to \beta_2,\ \alpha_1 \doteq F_1\alpha_2 \to \beta_1\}} \\
\begin{array}{cc}
\lambda x :: \langle\varnothing, \alpha_1 \wedge F_1\alpha_2 \to \beta_1\rangle \,/\, \{\alpha_1 \doteq F_1\alpha_2 \to \beta_1\} & F_2 :: \langle\{y \mapsto F_2\alpha_3\}, F_2\alpha_3\rangle \,/\, \varnothing \\
@^{\beta_1} :: \langle\{x \mapsto \alpha_1 \wedge F_1\alpha_2\}, \beta_1\rangle \,/\, \{\alpha_1 \doteq F_1\alpha_2 \to \beta_1\} & y^{\alpha_3} :: \langle\{y \mapsto \alpha_3\}, \alpha_3\rangle \,/\, \varnothing \\
x^{\alpha_1} :: \langle\{x \mapsto \alpha_1\}, \alpha_1\rangle \,/\, \varnothing \qquad F_1 :: \langle\{x \mapsto F_1\alpha_2\}, F_1\alpha_2\rangle \,/\, \varnothing & \\
x^{\alpha_2} :: \langle\{x \mapsto \alpha_2\}, \alpha_2\rangle \,/\, \varnothing &
\end{array}
\end{array}
$$

We then proceed to solve the collected constraints by β-unification, producing the substitution chain

$$\left\langle \begin{array}{l} \{[\alpha_1 := F_1\alpha_2 \to \beta_1]\}, \{[F_2 := \Box \wedge F_1\Box]\}, \{[|\alpha_3|_1 := F_1\alpha_2 \to \beta_2]\}, \\ \{[|\alpha_3|_2 := \alpha_2]\}, \{[\beta_1 := \beta_2]\} \end{array} \right\rangle$$

and by applying it to the analysis tree, we generate the following analysis tree which also qualifies as a typing derivation, because all constraint sets are solved (solved constraint sets are omitted):

- $@^{\beta_2} :: \langle \{y \mapsto (F_1\alpha_2 \to \beta_2) \wedge F_1\alpha_2\}, \beta_2 \rangle$
  - $\lambda x :: \langle \varnothing, (F_1\alpha_2 \to \beta_2) \wedge F_1\alpha_2 \to \beta_2 \rangle$
    - $@^{\beta_2} :: \left\langle \begin{array}{l} \{x \mapsto (F_1\alpha_2 \to \beta_2) \wedge F_1\alpha_2\}, \\ \beta_2 \end{array} \right\rangle$
      - $x^{(F_1\alpha_2 \to \beta_2)} :: \left\langle \begin{array}{l} \{x \mapsto F_1\alpha_2 \to \beta_2\}, \\ F_1\alpha_2 \to \beta_2 \end{array} \right\rangle$
      - $F_1 :: \langle \{x \mapsto F_1\alpha_2\}, F_1\alpha_2 \rangle$
        - $x^{\alpha_2} :: \langle \{x \mapsto \alpha_2\}, \alpha_2 \rangle$
  - $\wedge :: \langle \{y \mapsto (F_1\alpha_2 \to \beta_2) \wedge F_1\alpha_2\}, (F_1\alpha_2 \to \beta_2) \wedge F_1\alpha_2 \rangle$
    - $y^{(F_1\alpha_2 \to \beta_2)} :: \left\langle \begin{array}{l} \{y \mapsto F_1\alpha_2 \to \beta_2\}, \\ F_1\alpha_2 \to \beta_2 \end{array} \right\rangle$
    - $F_1 :: \left\langle \begin{array}{l} \{y \mapsto F_1\alpha_2\}, \\ F_1\alpha_2 \end{array} \right\rangle$
      - $y^{\alpha_2} :: \langle \{y \mapsto \alpha_2\}, \alpha_2 \rangle$

2.2.2 Compositional Analysis with Eager Substitutions

As System $\mathbb{I}$ has principal typings, we can choose instead to completely solve type constraints as soon as they arise in the process of building the analysis tree. Furthermore, we can apply the substitutions solving the constraints to the analysis trees. This means that at every point, an analysis tree generated so far will also be a valid typing derivation. This strategy can also be used in inferring types for terms in the simply-typed λ-calculus, but cannot be adapted (or easily so) to the Hindley/Milner type system, because typings of that system are insufficient for representing intermediate inference results for bottom-up inference [17].

Inference will proceed as before until we reach the point where a constraint is first produced:

- $\boxed{@^{\beta_1} :: \langle \{x \mapsto \alpha_1 \wedge F_1\alpha_2\}, \beta_1 \rangle \,/\, \{\alpha_1 \doteq F_1\alpha_2 \to \beta_1\}}$
  - $x^{\alpha_1} :: \langle \{x \mapsto \alpha_1\}, \alpha_1 \rangle$
  - $F_1 :: \langle \{x \mapsto F_1\alpha_2\}, F_1\alpha_2 \rangle$
    - $x^{\alpha_2} :: \langle \{x \mapsto \alpha_2\}, \alpha_2 \rangle$
- $y^{\alpha_3} :: \langle \{y \mapsto \alpha_3\}, \alpha_3 \rangle$

As before, we remove solved constraints from constraint sets and omit empty constraint sets. We can immediately solve the constraint, generating the substitution chain $\langle\{[\![\alpha_1 := F_1\alpha_2 \to \beta_1]\!]\}\rangle$. By applying it to the analysis tree, we obtain the following typing derivations (modified nodes are enclosed in dashed boxes):

$$
\begin{array}{ccc}
 & \boxed{@^{\beta_1} :: \langle\{x \mapsto (F_1\alpha_2 \to \beta_1) \wedge F_1\alpha_2\}, \beta_1\rangle} & \qquad y^{\alpha_3} :: \langle\{y \mapsto \alpha_3\}, \alpha_3\rangle \\
\boxed{x^{(F_1\alpha_2 \to \beta_1)} :: \left\langle \begin{array}{l} \{x \mapsto F_1\alpha_2 \to \beta_1\}, \\ F_1\alpha_2 \to \beta_1 \end{array} \right\rangle} & F_1 :: \langle\{x \mapsto F_1\alpha_2\}, F_1\alpha_2\rangle & \\
 & x^{\alpha_2} :: \langle\{x \mapsto \alpha_2\}, \alpha_2\rangle &
\end{array}
$$

Similarly, we can repeat the reasoning in section 2.2.1 to reach the next step where a constraint arises:

$$
\begin{array}{c}
\boxed{@^{\beta_2} :: \langle\{y \mapsto F_2\alpha_3\}, \beta_2\rangle \,/\, \{\alpha_1 \wedge F_1\alpha_2 \to \beta_1 \doteq F_2\alpha_3 \to \beta_2\}} \\
\lambda x :: \langle\varnothing, (F_1\alpha_2 \to \beta_1) \wedge F_1\alpha_2 \to \beta_1\rangle \qquad F_2 :: \langle\{y \mapsto F_2\alpha_3\}, F_2\alpha_3\rangle \\
@^{\beta_1} :: \langle\{x \mapsto (F_1\alpha_2 \to \beta_1) \wedge F_1\alpha_2\}, \beta_1\rangle \qquad y^{\alpha_3} :: \langle\{y \mapsto \alpha_3\}, \alpha_3\rangle \\
x^{(F_1\alpha_2 \to \beta_1)} :: \left\langle \begin{array}{l} \{x \mapsto F_1\alpha_2 \to \beta_1\}, \\ F_1\alpha_2 \to \beta_1 \end{array} \right\rangle \qquad F_1 :: \langle\{x \mapsto F_1\alpha_2\}, F_1\alpha_2\rangle \\
x^{\alpha_2} :: \langle\{x \mapsto \alpha_2\}, \alpha_2\rangle
\end{array}
$$

When the constraint is solved and the resulting substitution applied to the analysis tree, we obtain the following typing derivation, identical to the one obtained at the end of the previous subsection:

$$
\begin{array}{c}
\boxed{@^{\beta_2} :: \langle\{y \mapsto (F_1\alpha_2 \to \beta_2) \wedge F_1\alpha_2\}, \beta_2\rangle} \\
\boxed{\lambda x :: \langle\varnothing, (F_1\alpha_2 \to \beta_2) \wedge F_1\alpha_2 \to \beta_2\rangle} \quad \boxed{\wedge :: \langle\{y \mapsto (F_1\alpha_2 \to \beta_2) \wedge F_1\alpha_2\}, (F_1\alpha_2 \to \beta_2) \wedge F_1\alpha_2\rangle} \\
\boxed{@^{\beta_2} :: \left\langle \begin{array}{l} \{x \mapsto (F_1\alpha_2 \to \beta_2) \wedge F_1\alpha_2\}, \\ \beta_2 \end{array} \right\rangle} \quad \boxed{y^{(F_1\alpha_2 \to \beta_2)} :: \left\langle \begin{array}{l} \{y \mapsto F_1\alpha_2 \to \beta_2\}, \\ F_1\alpha_2 \to \beta_2 \end{array} \right\rangle} \quad \boxed{F_1 :: \left\langle \begin{array}{l} \{y \mapsto F_1\alpha_2\}, \\ F_1\alpha_2 \end{array} \right\rangle} \\
\boxed{x^{(F_1\alpha_2 \to \beta_2)} :: \left\langle \begin{array}{l} \{x \mapsto F_1\alpha_2 \to \beta_2\}, \\ F_1\alpha_2 \to \beta_2 \end{array} \right\rangle} \quad F_1 :: \langle\{x \mapsto F_1\alpha_2\}, F_1\alpha_2\rangle \qquad \boxed{y^{\alpha_2} :: \langle\{y \mapsto \alpha_2\}, \alpha_2\rangle} \\
x^{\alpha_2} :: \langle\{x \mapsto \alpha_2\}, \alpha_2\rangle
\end{array}
$$

This approach is compositional. It may not be optimal for some applications. In this approach, constraints are immediately solved and the resulting substitutions are immediately applied to the entirety of both subtrees of the application. In an implementation, this may involve destructively modifying the subtrees or creating new subtrees and discarding the old ones. Suppose we wish to edit the program by changing some node, e.g., changing a λx to a λy. This may potentially require reanalyzing the entire program. The change from λx to λy may imply a change in the solution of some constraint generated closer to the root of the program, perhaps at the very root. This may in turn imply a change in a substitution applied to the entirety of the subtrees of the node generating the constraint. At this point, all of the analysis data in the analysis tree may be invalid and may need to be thrown out and regenerated from scratch. So this approach has problems doing incremental reanalysis after changes.

2.2.3 Compositional and Incremental Analysis with Lazy Substitutions

The alternative is to solve constraints as they arise, just as in the eager compositional analysis of the previous subsection, but instead of immediately applying the resulting substitutions, we collect and remember them, effectively composing them incrementally.

The analysis forest at the point where a first constraint is introduced, namely:

$$
\begin{array}{ccc}
\multicolumn{2}{c}{\boxed{@^{\beta_1} :: \langle \{x \mapsto \alpha_1 \wedge F_1\alpha_2\}, \beta_1\rangle \,/\, \{\alpha_1 \doteq F_1\alpha_2 \to \beta_1\}}} & y^{\alpha_3} :: \langle \{y \mapsto \alpha_3\}, \alpha_3\rangle \\
\diagup & \diagdown & \\
x^{\alpha_1} :: \langle \{x \mapsto \alpha_1\}, \alpha_1\rangle & F_1 :: \langle \{x \mapsto F_1\alpha_2\}, F_1\alpha_2\rangle & \\
 & | & \\
 & x^{\alpha_2} :: \langle \{x \mapsto \alpha_2\}, \alpha_2\rangle &
\end{array}
$$

is changed to

$$
\begin{array}{ccc}
\multicolumn{2}{c}{\boxed{\langle \{\![\alpha_1 := F_1\alpha_2 \to \beta_1]\!\}\rangle :: \langle \{x \mapsto (F_1\alpha_2 \to \beta_1) \wedge F_1\alpha_2\}, \beta_1\rangle}} & y^{\alpha_3} :: \langle \{y \mapsto \alpha_3\}, \alpha_3\rangle \\
\multicolumn{2}{c}{|} & \\
\multicolumn{2}{c}{@^{\beta_1} :: \langle \{x \mapsto \alpha_1 \wedge F_1\alpha_2\}, \beta_1\rangle / \{\alpha_1 \doteq F_1\alpha_2 \to \beta_1\}} & \\
\diagup & \diagdown & \\
x^{\alpha_1} :: \langle \{x \mapsto \alpha_1\}, \alpha_1\rangle & F_1 :: \langle \{x \mapsto F_1\alpha_2\}, F_1\alpha_2\rangle & \\
 & | & \\
 & x^{\alpha_2} :: \langle \{x \mapsto \alpha_2\}, \alpha_2\rangle &
\end{array}
$$

where a new node using the *substitution rule* with the substitution chain $\langle \{\![\alpha_1 := F_1\alpha_2 \to \beta_1]\!\}\rangle$ is added to the forest, and again all empty constraint sets are omitted throughout the analysis forest. Although the substitution rule is admissible using the other typing rules, it is convenient to have it as an explicit rule in order to put suspended substitutions into analysis trees. At the point where a second constraint is introduced, namely:

$$
\begin{array}{c}
\boxed{@^{\beta_2} :: \langle \{y \mapsto F_2\alpha_3\}, \beta_2\rangle / \{(F_1\alpha_2 \to \beta_1) \wedge F_1\alpha_2 \to \beta_1 \doteq F_2\alpha_3 \to \beta_2\}} \\
\begin{array}{cc}
\begin{array}{c}
\lambda x :: \langle \varnothing, (F_1\alpha_2 \to \beta_1) \wedge F_1\alpha_2 \to \beta_1\rangle \\
| \\
\langle \{\![\alpha_1 := F_1\alpha_2 \to \beta_1]\!\}\rangle :: \langle \{x \mapsto (F_1\alpha_2 \to \beta_1) \wedge F_1\alpha_2\}, \beta_1\rangle \\
| \\
@^{\beta_1} :: \langle \{x \mapsto \alpha_1 \wedge F_1\alpha_2\}, \beta_1\rangle / \{\alpha_1 \doteq F_1\alpha_2 \to \beta_1\} \\
\begin{array}{cc}
x^{\alpha_1} :: \langle \{x \mapsto \alpha_1\}, \alpha_1\rangle &
\begin{array}{c}
F_1 :: \langle \{x \mapsto F_1\alpha_2\}, F_1\alpha_2\rangle \\
| \\
x^{\alpha_2} :: \langle \{x \mapsto \alpha_2\}, \alpha_2\rangle
\end{array}
\end{array}
\end{array}
&
\begin{array}{c}
F_2 :: \langle \{y \mapsto F_2\alpha_3\}, F_2\alpha_3\rangle \\
| \\
y^{\alpha_3} :: \langle \{y \mapsto \alpha_3\}, \alpha_3\rangle
\end{array}
\end{array}
\end{array}
$$

solving the constraint yields the substitution chain

$$C = \langle \{\![F_2 := \square \wedge F_1\square]\!\}\,, \{\![|\alpha_3|_1 := F_1\alpha_2 \to \beta_2]\!\}\,, \{\![|\alpha_3|_2 := \alpha_2]\!\}\,, \{\![\beta_1 := \beta_2]\!\}\rangle$$

and thus the resulting analysis tree:

$$
\begin{array}{c}
\boxed{C :: \langle \{y \mapsto (F_1\alpha_2 \to \beta_2) \wedge F_1\alpha_2\}, \beta_2\rangle} \\
| \\
@^{\beta_2} :: \langle \{y \mapsto F_2\alpha_3\}, \beta_2\rangle / \{(F_1\alpha_2 \to \beta_1) \wedge F_1\alpha_2 \to \beta_1 \doteq F_2\alpha_3 \to \beta_2\} \\
\begin{array}{cc}
\begin{array}{c}
\lambda x :: \langle \varnothing, (F_1\alpha_2 \to \beta_1) \wedge F_1\alpha_2 \to \beta_1\rangle \\
| \\
\langle \{\![\alpha_1 := F_1\alpha_2 \to \beta_1]\!\}\rangle :: \langle \{x \mapsto (F_1\alpha_2 \to \beta_1) \wedge F_1\alpha_2\}, \beta_1\rangle \\
| \\
@^{\beta_1} :: \langle \{x \mapsto \alpha_1 \wedge F_1\alpha_2\}, \beta_1\rangle / \{\alpha_1 \doteq F_1\alpha_2 \to \beta_1\} \\
\begin{array}{cc}
x^{\alpha_1} :: \langle \{x \mapsto \alpha_1\}, \alpha_1\rangle &
\begin{array}{c}
F_1 :: \langle \{x \mapsto F_1\alpha_2\}, F_1\alpha_2\rangle \\
| \\
x^{\alpha_2} :: \langle \{x \mapsto \alpha_2\}, \alpha_2\rangle
\end{array}
\end{array}
\end{array}
&
\begin{array}{c}
F_2 :: \langle \{y \mapsto F_2\alpha_3\}, F_2\alpha_3\rangle \\
| \\
y^{\alpha_3} :: \langle \{y \mapsto \alpha_3\}, \alpha_3\rangle
\end{array}
\end{array}
\end{array}
$$

2.2.4 Example of Incremental Reanalysis

As discussed earlier, the observationally equivalent term ($\mathsf{let}\ x = y\ \mathsf{in}\ xx$) can be typed in the Hindley/Milner type system by algorithm $\mathcal{W}$, but this is done in a non-compositional way. We can imagine trying to create a variant of $\mathcal{W}$ for incremental reanalysis, but it would still need to reanalyze the body e' in ($\mathsf{let}\ x = e\ \mathsf{in}\ e'$) when the definition e changes. To illustrate that System $\mathbb{I}$ does not have this problem, we show how changing the argument of the application in our example term does not require us to reanalyze the function which consumes it.

We start with the completed analysis tree of section 2.2.3 from just above. Then we change the argument (i.e., the definition of x) from y to $\lambda z.y$, and analyze the new argument

$$\begin{array}{ccc}
 & \lambda x :: \langle \varnothing, (F_1\alpha_2 \to \beta_1) \wedge F_1\alpha_2 \to \beta_1 \rangle & \\
 & | & \\
 & \langle \{\!\![\alpha_1 := F_1\alpha_2 \to \beta_1]\!\!\} \rangle :: \langle \{x \mapsto (F_1\alpha_2 \to \beta_1) \wedge F_1\alpha_2\}, \beta_1 \rangle & \\
 & | & \\
 & @^{\beta_1} :: \langle \{x \mapsto \alpha_1 \wedge F_1\alpha_2\}, \beta_1 \rangle / \{\alpha_1 \doteq F_1\alpha_2 \to \beta_1\} & \\
 \diagup & & \diagdown \\
 x^{\alpha_1} :: \langle \{x \mapsto \alpha_1\}, \alpha_1 \rangle & & F_1 :: \langle \{x \mapsto F_1\alpha_2\}, F_1\alpha_2 \rangle \\
 & & | \\
 & & x^{\alpha_2} :: \langle \{x \mapsto \alpha_2\}, \alpha_2 \rangle
\end{array}
\qquad
\begin{array}{c}
\boxed{F_2 :: \langle \{y \mapsto F_2\alpha_3\}, F_2(\alpha_4 \to \alpha_3) \rangle} \\
\vdots \\
\boxed{\lambda z^{\alpha_4} :: \langle \{y \mapsto \alpha_3\}, \alpha_4 \to \alpha_3 \rangle} \\
\vdots \\
\boxed{y^{\alpha_3} :: \langle \{y \mapsto \alpha_3\}, \alpha_3 \rangle}
\end{array}$$

and finally combine the two analyses under the application

$$\begin{array}{ccc}
 & \boxed{@^{\beta_2} :: \langle \{y \mapsto F_2\alpha_3\}, \beta_2 \rangle / \{(F_1\alpha_2 \to \beta_1) \wedge F_1\alpha_2 \to \beta_1 \doteq F_2(\alpha_4 \to \alpha_3) \to \beta_2\}} & \\
 \diagup & & \diagdown \\
 \lambda x :: \langle \varnothing, (F_1\alpha_2 \to \beta_1) \wedge F_1\alpha_2 \to \beta_1 \rangle & & F_2 :: \langle \{y \mapsto F_2\alpha_3\}, F_2(\alpha_4 \to \alpha_3) \rangle \\
 | & & | \\
 \langle \{\!\![\alpha_1 := F_1\alpha_2 \to \beta_1]\!\!\} \rangle :: \langle \{x \mapsto (F_1\alpha_2 \to \beta_1) \wedge F_1\alpha_2\}, \beta_1 \rangle & & \lambda z^{\alpha_4} :: \langle \{y \mapsto \alpha_3\}, \alpha_4 \to \alpha_3 \rangle \\
 | & & | \\
 @^{\beta_1} :: \langle \{x \mapsto \alpha_1 \wedge F_1\alpha_2\}, \beta_1 \rangle / \{\alpha_1 \doteq F_1\alpha_2 \to \beta_1\} & & y^{\alpha_3} :: \langle \{y \mapsto \alpha_3\}, \alpha_3 \rangle \\
 \diagup \quad \diagdown & & \\
 x^{\alpha_1} :: \langle \{x \mapsto \alpha_1\}, \alpha_1 \rangle \quad F_1 :: \langle \{x \mapsto F_1\alpha_2\}, F_1\alpha_2 \rangle & & \\
 | & & \\
 x^{\alpha_2} :: \langle \{x \mapsto \alpha_2\}, \alpha_2 \rangle & &
\end{array}$$

And solve the constraint as before. The important thing to observe is that the entire analysis subtree for $(\lambda x.xx)$ is reused without any change.

2.3 Remarks

One may fall into the trap of believing that we advocate one of these strategies as being the "best". The approach that is best is highly dependent on the application for which it is intended. The lazy incremental analysis is probably the best for real-time analysis in an integrated development environment, whereas one could potentially imagine using the eager compositional analysis on binary objects that will only later be later composed to form a complete program. The traditional bottom-up analysis can always be used for batch program analysis as appropriate. We believe the strength lies not in any one of these strategies, but the fact that a single framework supports the entire gamut of possibilities.

3 Implementing System $\mathbb{I}$

Here is presented a new, more streamlined definition of System $\mathbb{I}$ and the finite rank β-unification algorithm intended to be used as a guide towards implementation. The definitions of terms, types, and expansions were covered earlier in section 2.1.

3.1 Variables

Term variables are members of the countably infinite set λ-Var. Let x, y, and z range over λ-Var. Let $\mathsf{FV}(M)$ be the free term variables of the λ-term M. Type variables, also called T-variables, are members of the countably infinite set TVar. Let α, β, and γ range over TVar. Expansion variables, also called E-variables, are members of the countably infinite set EVar. Let F, G, and H range over EVar. Let $\mathsf{Var} = \mathsf{TVar} \cup \mathsf{EVar}$ (all the variables which can occur in types). Let $\mathsf{var}(X)$ be the set of all type or expansion variables which occur in X, whatever X is.

3.2 Renaming of Variables in Types

In previous descriptions of System $\mathbb{I}$, such as in [9] and [11], the variable-renaming mechanism required as part of substituting into expansions was a very complex process. While there is presently active research into developing an equivalent form of substitution which is independent of variable-renaming, we present here a simpler form which only depends on variable-renaming in a generic way, i.e., it does not require a commitment to a specific variable-renaming mechanism. This is partially based on unpublished work joint with Yates [19].

A variable-renaming function is denoted $|\ |_i$ where i is a positive integer, and the result of applying it to $v \in \mathsf{Var}$ is denoted $|v|_i$. We use $\boldsymbol{m}$ and $\boldsymbol{n}$ to denote sequences, possibly empty, of positive integers. If $\boldsymbol{n}$ is the sequence of positive integers $i_1, i_2, \ldots, i_k$ and $v \in \mathsf{Var}$, we write $|v|_{\boldsymbol{n}}$ as an abbreviation for $|\cdots||v|_{i_1}|_{i_2}\cdots|_{i_k}$. If $\boldsymbol{n}$ is the empty sequence of positive integers, then $|v|_{\boldsymbol{n}} = v$.

We assume the existence of a countably infinite family of variable-renaming functions $|\ |_i$, one for every $i \geq 1$, satisfying the properties:

(i) For all $v, w \in \mathsf{Var}$ and all sequences $\boldsymbol{m}, \boldsymbol{n}$ of positive integers, if $|v|_{\boldsymbol{m}} = |w|_{\boldsymbol{n}}$ then $v = w$ and $\boldsymbol{m} = \boldsymbol{n}$.

(ii) There are countably infinite subsets $\mathsf{TVar}_b \subset \mathsf{TVar}$ and $\mathsf{EVar}_b \subset \mathsf{EVar}$ such that for every $v \in \mathsf{TVar}_b \cup \mathsf{EVar}_b$ and every $i \geq 1$ it is the case that $v \neq |v|_i$.

There are infinitely many ways of defining variable-renaming functions that satisfy these two properties.

For later reference, we call the sets TVar_b and EVar_b in the second property above the sets of *basic T-variables* and *basic E-variables*, respectively. Let $\mathsf{Var}_b = \mathsf{TVar}_b \cup \mathsf{EVar}_b$. We call variable w a *descendant* of variable v if $|v|_{\boldsymbol{n}} = w$ for some sequence $\boldsymbol{n}$ of positive integers; because $\boldsymbol{n}$ can be the empty sequence, v is a descendant of itself as a special case. If X is an object containing T-variables and E-variables, we define $\mathsf{var}_b(X)$ as follows:

$$\mathsf{var}_b(X) = \{\, v \in \mathsf{Var}_b \mid \text{ there is } w \in \mathsf{var}(X) \text{ such that } w \text{ is a descendant of } v \,\}.$$

For theoretical purposes, in order to make the application of substitutions to types (defined below) a function, we assume the variable-renaming functions to be predetermined and fixed. This is consistent with an implementation which does not fix them in advance but which remembers all of its choices. In practice this proves much easier to implement than the approach (based on *offsets*) used in previous presentations. One method of implementing such a family of variable-renaming functions is to represent the functions as finite maps, allocating them as necessary. When a renaming function is applied to a variable $v \in \mathsf{Var}$, it looks up v in the map: If v already has a mapping that mapping is used; if v does not already have a mapping, simply generate and return a fresh variable, storing it in the map for future reference.

A variable-renaming function $|\cdot|_i : \mathsf{Var} \to \mathsf{Var}$ is lifted to a function $\overline{|\cdot|}_i : \mathsf{Type} \to \mathsf{Type}$ in the obvious way:

(i) $\overline{|\alpha|}_i = |\alpha|_i$.

(ii) $\overline{|\tau \to \bar{\tau}|}_i = \overline{|\tau|}_i \to \overline{|\bar{\tau}|}_i$.

(iii) $\overline{|\tau_1 \wedge \tau_2|}_i = \overline{|\tau_1|}_i \wedge \overline{|\tau_2|}_i$.

(iv) $\overline{|F\,\tau|}_i = |F|_i\,\overline{|\tau|}_i$.

3.3 Substitutions on Types

A *substitution* is a total function $S : \mathsf{Var} \to (\mathsf{Expansion} \cup \mathsf{Type}^{\to})$ which respects sorts, i.e., $S(F) \in \mathsf{Expansion}$ for every $F \in \mathsf{EVar}$ and $S(\alpha) \in \mathsf{Type}^{\to}$ for every $\alpha \in \mathsf{TVar}$. A substitution S acts *trivially* on a type variable α iff $S(\alpha) = \alpha$ and on a expansion variable F iff $S(F) = F\Box$. A *small substitution* is a substitution that acts non-trivially on at most one variable. The notation $\{\![v := X]\!\}$ denotes the small substitution which maps v to X and is trivial elsewhere. A substitution S is lifted to a function $\widetilde{S}$ from Type to Type as follows:

(i) $\widetilde{S}(\alpha) = S(\alpha)$.

(ii) $\widetilde{S}(\tau \to \bar{\tau}) = \widetilde{S}(\tau) \to \widetilde{S}(\bar{\tau})$.

(iii) $\widetilde{S}(\tau_1 \wedge \tau_2) = \widetilde{S}(\tau_1) \wedge \widetilde{S}(\tau_2)$.

(iv) $\widetilde{S}(F\,\tau) = e[\widetilde{S}(\overline{|\tau|}_1), \ldots, \widetilde{S}(\overline{|\tau|}_n)]$ where $e = S(F)$ has $n \geq 1$ holes.

A *substitution chain* C is a finite sequence of small substitutions, written in the form $\langle S_1, \ldots, S_n \rangle$. Given an object X (a type, or as defined later, a skeleton), the application of the chain $C = \langle S_1, \ldots, S_n \rangle$ to X, written $C(X)$, is defined as $\widetilde{S}_n(\cdots \widetilde{S}_2(\widetilde{S}_1(X)) \cdots)$.

3.4 Type Constraint Sets

A *type constraint* is a pair of types written in the form $(\tau \doteq \tau')$. The order of the pairs is significant and the two types must not be switched. The left side of the constraint is considered to be a positive position while the right side is negative; this fact is not needed to understand this paper. A constraint $(\tau \doteq \tau')$ is *solved* iff $\tau = \tau'$. Given a substitution chain C and a constraint $(\tau \doteq \tau')$, let $C(\tau \doteq \tau') = (C(\tau) \doteq C(\tau'))$. Given an expansion variable F and a constraint $(\tau \doteq \tau')$, let $F(\tau \doteq \tau') = (F\tau \doteq F\tau')$.

A *type constraint set* Δ is a set of constraints. Let Δ range over constraint sets. A constraint set is solved iff all of its constraints are solved. Given a substitution chain C and a constraint set Δ, let $C(\Delta) = \{\, C(\tau \doteq \tau') \mid (\tau \doteq \tau') \in \Delta \,\}$. Given an expansion variable F and a constraint set Δ, make the definition that $F(\Delta) = \{\, F(\tau \doteq \tau') \mid (\tau \doteq \tau') \in \Delta \,\}$. A substitution chain C is a *solution* of a constraint set Δ iff $C(\Delta)$ is solved.

3.5 Beta-Unification

The set Δ of constraints constructed in the course of generating the skeleton of a term M is an instance of *β-unification*. It is undecidable whether an arbitrary instance of β-unification has a solution. The constraint set Δ induced by a term M satisfies several restrictions that makes it better behaved than arbitrary instances of β-unification. These restrictions and the reasons why they are important are not discussed here. If an implementer follows the definitions in this paper, then the restrictions will hold.

We design a non-deterministic rewrite algorithm to find solutions to appropriately restricted constraint sets, in particular, those induced by terms of the pure λ-calculus. This algorithm cannot be applied to arbitrary constraint sets.

The operation of our algorithm is based on the rewrite rules shown in figure 1. The presentation is self-contained. A *rewrite step* is in one of 4 possible forms, for some constraint sets Δ_0 and Δ_1:

- $\Delta_0 \underset{\mathsf{init}}{\Longrightarrow} \Delta_1$, application of $\mathsf{simplify}(\ )$ to Δ_0 to obtain Δ_1.
- $\Delta_0 \overset{S}{\underset{+\mathsf{T}}{\Longrightarrow}} \Delta_1$, elimination of a T-variable which has a positive occurrence in Δ_0.
- $\Delta_0 \overset{S}{\underset{-\mathsf{T}}{\Longrightarrow}} \Delta_1$, elimination of a T-variable which has a negative occurrence in Δ_0.
- $\Delta_0 \overset{S}{\underset{\mathsf{E}}{\Longrightarrow}} \Delta_1$, elimination of an E-variable which has a positive occurrence in Δ_0.

In fact, each of the last 3 steps above also includes an application of $\mathsf{simplify}(\ )$. Thus a rewrite step of the form $\Delta_0 \underset{\mathsf{init}}{\Longrightarrow} \Delta_1$ needs to be used only once initially, in case $\Delta_0 \neq \mathsf{simplify}(\Delta_0)$.

Mode of operation:

- Initial step: $\Delta \underset{\mathsf{init}}{\Longrightarrow} \mathsf{simplify}(\Delta)$.

- $\Delta_0 \overset{S}{\underset{r}{\Longrightarrow}} \Delta_1$, provided:
  - $\cdot\ \Delta_0 = \Delta \cup \boldsymbol{F}\{\tau \doteq \tau'\}$,
  - $\cdot\ \tau \doteq \tau' \Rightarrow S$ is an instance of (**rule** r) for $r \in \{\, +\mathsf{T},\ -\mathsf{T},\ \mathsf{E}\,\}$,
  - $\cdot\ \Delta_1 = \mathsf{simplify}(S\Delta_0)$.

Rewrite rules:

$$\begin{array}{llll}
\alpha \doteq \bar{\tau} & \Rightarrow & \{\![\alpha := \bar{\tau}]\!\} & (\textbf{rule}\ \ +\mathsf{T}) \\
\bar{\tau} \doteq \alpha & \Rightarrow & \{\![\alpha := \bar{\tau}]\!\} & (\textbf{rule}\ \ -\mathsf{T}) \\
F\bar{\tau} \doteq e[\bar{\tau}_1, \ldots, \bar{\tau}_n] & \Rightarrow & \{\![F := e]\!\} \quad \text{where } e \neq F\Box & (\textbf{rule}\ \ \mathsf{E})
\end{array}$$

Simplifying constraint sets:

- $\mathsf{simplify}(\varnothing) = \varnothing$.

- $\mathsf{simplify}(\{\tau \doteq \tau'\} \cup \Delta) = \mathsf{simplify}(\tau \doteq \tau') \cup \mathsf{simplify}(\Delta)$.

- $$\mathsf{simplify}(\tau \doteq \tau') = \begin{cases}
F\,\mathsf{simplify}(\tau_1 \doteq \tau_1') & \text{if } \tau = F\tau_1 \\
 & \text{and } \tau' = F\tau_1', \\
\mathsf{simplify}(\tau_1' \doteq \tau_1) \cup \mathsf{simplify}(\tau_2 \doteq \tau_2') & \text{if } \tau = \tau_1 \to \tau_2 \\
 & \text{and } \tau' = \tau_1' \to \tau_2', \\
\mathsf{simplify}(\tau_1 \doteq \tau_1') \cup \mathsf{simplify}(\tau_2 \doteq \tau_2') & \text{if } \tau = \tau_1 \wedge \tau_2 \\
 & \text{and } \tau' = \tau_1' \wedge \tau_2', \\
\varnothing & \text{if } \tau = \tau', \\
\{\tau \doteq \tau'\} & \text{otherwise.}
\end{cases}$$

Fig. 1. Constraint set rewriting algorithm (a modification of algorithm Unify in [11]).

Let the partial function β-unify from constraint sets to substitution chains be defined as follows. If there is at least one sequence of rewrite steps such that

$$\Delta \underset{\mathsf{init}}{\Longrightarrow} \Delta_1 \overset{S_1}{\underset{r_1}{\Longrightarrow}} \Delta_2 \overset{S_2}{\underset{r_2}{\Longrightarrow}} \cdots \overset{S_n}{\underset{r_n}{\Longrightarrow}} \Delta_n$$

and such that $\Delta_n = \varnothing$, then let $\beta\text{-}\mathsf{unify}(\Delta) = \langle S_1, S_2, \ldots, S_n \rangle$ for exactly one such sequence (chosen arbitrarily). Otherwise, let $\beta\text{-}\mathsf{unify}(\Delta)$ be undefined. An instance Δ of β-unification *succeeds* iff $\beta\text{-}\mathsf{unify}(\Delta) = C$ for some chain C, and in this case, C is a solution for Δ. As a function from types to types, C behaves effectively as $\widetilde{S}$ for some large substitution S, but this fact is neither straightforward to establish nor is it necessary.

3.6 Type Environments

Type environments were introduced informally in section 2. Formally, a type environment A is a partial function from λ-Var to the set Type of types, with finite domain. Functions are viewed as sets of pairs, so if the domain of definition of A is $\mathsf{dom}(A) = \{x_1, \dots, x_n\}$, A can be written in the form $A = \{x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n\}$ for some $\tau_1, \dots, \tau_n \in \mathsf{Type}$. This means $A(x_i) = \tau_i$ for every $1 \leq i \leq n$ and $A(y)$ is undefined for $y \notin \{x_1, \dots, x_n\}$. We need the following operations on type environments, where $F \in \mathsf{EVar}$ and A and B are arbitrary type environments:

$$
\begin{array}{lcl}
F\,A & = & \{\, x \mapsto F\tau \,|\, A(x) = \tau \,\}, \\
A \wedge B & = & \{\, x \mapsto \tau_1 \wedge \tau_2 \,|\, A(x) = \tau_1,\ B(x) = \tau_2 \,\} \ \cup \\
 & & \{\, x \mapsto \tau \,|\, A(x) = \tau,\ x \notin \mathsf{dom}(B) \,\} \ \cup \\
 & & \{\, x \mapsto \tau \,|\, B(x) = \tau,\ x \notin \mathsf{dom}(A) \,\}, \\
A_x & = & \{\, y \mapsto \tau \,|\, A(y) = \tau,\ x \neq y \,\}.
\end{array}
$$

Note that the intersection type constructor ("$\wedge$") is neither associative nor commutative in types.

3.7 Skeletons and Typing Rules

A *skeleton* is a term representing in a compact way all of the essential information in a derivation using the typing rules. They are given by the following pseudo-grammar:

$$\mathcal{Q} ::= x^{\bar\tau} \mid \mathcal{Q}_1 @^{\bar\tau} \mathcal{Q}_2 \mid F\mathcal{Q} \mid \lambda x.\mathcal{Q} \mid \lambda x^{\bar\tau}.\mathcal{Q} \mid \mathcal{Q}_1 \wedge \mathcal{Q}_2 \mid \langle C, \mathcal{Q} \rangle$$

$$\frac{}{x \Rightarrow x^{\bar\tau} : \langle \{x \mapsto \bar\tau\}, \bar\tau \rangle \,/ \varnothing}\ (x^{\bar\tau}) \qquad \frac{M \Rightarrow \mathcal{Q} : \langle A, \tau \rangle \,/ \Delta}{M \Rightarrow F\mathcal{Q} : \langle FA, F\tau \rangle \,/ F\Delta}\ (F)$$

$$\frac{M \Rightarrow \mathcal{Q} : \langle A \cup \{x \mapsto \tau\}, \bar\tau \rangle \,/ \Delta}{\lambda x.M \Rightarrow \lambda x.\mathcal{Q} : \langle A_x, \tau \to \bar\tau \rangle \,/ \Delta}\ (\lambda x) \qquad \frac{M \Rightarrow \mathcal{Q} : \langle A, \bar\tau' \rangle \,/ \Delta; \quad x \notin \mathsf{dom}(A)}{\lambda x.M \Rightarrow \lambda x^{\bar\tau}.\mathcal{Q} : \langle A, \bar\tau \to \bar\tau' \rangle \,/ \Delta}\ (\lambda x^{\bar\tau})$$

$$\frac{M \Rightarrow \mathcal{Q}_1 : \langle A, \bar\tau' \rangle \,/ \Delta_1; \quad N \Rightarrow \mathcal{Q}_2 : \langle B, \tau \rangle \,/ \Delta_2}{MN \Rightarrow \mathcal{Q}_1 @^{\bar\tau} \mathcal{Q}_2 : \langle A \wedge B, \bar\tau \rangle \,/ \Delta_1 \cup \Delta_2 \cup \{\bar\tau' \doteq \tau \to \bar\tau\}}\ (@^{\bar\tau})$$

$$\frac{M \Rightarrow \mathcal{Q}_1 : \langle A, \tau_1 \rangle \,/ \Delta_1; \quad M \Rightarrow \mathcal{Q}_2 : \langle B, \tau_2 \rangle \,/ \Delta_2}{M \Rightarrow \mathcal{Q}_1 \wedge \mathcal{Q}_2 : \langle A \wedge B, \tau_1 \wedge \tau_2 \rangle \,/ \Delta_1 \cup \Delta_2}\ \wedge$$

Fig. 2. Typing rules.

The typing rules given in figure 2 derive judgements of the form $M \Rightarrow \mathcal{Q} : \langle A, \tau \rangle / \Delta$ which should be read as stating that "the term M has a corresponding skeleton $\mathcal{Q}$ which determines the final typing $\langle A, \tau \rangle$ and the constraints Δ". For each skeleton $\mathcal{Q}$, there is at most one such λ-term M, which is called the *term of the skeleton*. Note that it is always possible to find a skeleton, final typing, and constraint set for a λ-term, although the constraint set may not be solvable. A skeleton $\mathcal{Q}$ is *valid* iff a judgement $M \Rightarrow \mathcal{Q} : \langle A, \tau \rangle / \Delta$ can be derived. Henceforth, only valid skeletons are considered.

Each skeleton and its corresponding λ-term implicitly and automatically determines via the typing rules a final typing and a constraint set. If each constraint in the set is already solved (i.e., the constrained pair is already equal), then the skeleton is also called a *typing derivation for its term* and the final typing is *valid for the skeleton's term.* By convention, solved constraints are omitted when constraint sets are written. Furthermore, solved constraint sets may be optionally omitted together with the preceding "/". The constraints of a given skeleton $\mathcal{Q}$ may or may not be solvable. If they are solvable, the solution may be applied to the skeleton $\mathcal{Q}$ to produce another skeleton that is also a typing derivation.

Applying a lifted renaming to a skeleton is defined as follows:

(i) $\overline{|x^{\bar{\tau}}|}_i = x^{\overline{|\bar{\tau}|}_i}$.

(ii) $\overline{|\mathcal{Q}_1 @^{\bar{\tau}} \mathcal{Q}_2|}_i = \overline{|\mathcal{Q}_1|}_i @^{\overline{|\bar{\tau}|}_i} \overline{|\mathcal{Q}_2|}_i$.

(iii) $\overline{|F\mathcal{Q}|}_i = |F|_i \overline{|\mathcal{Q}|}_i$.

(iv) $\overline{|\lambda x.\mathcal{Q}|}_i = \lambda x.\overline{|\mathcal{Q}|}_i$.

(v) $\overline{|\lambda x^{\bar{\tau}}.\mathcal{Q}|}_i = \lambda x^{\overline{|\bar{\tau}|}_i}.\overline{|\mathcal{Q}|}_i$.

(vi) $\overline{|\mathcal{Q}_1 \wedge \mathcal{Q}_2|}_i = \overline{|\mathcal{Q}_1|}_i \wedge \overline{|\mathcal{Q}_2|}_i$.

(vii) $\overline{|\langle C, \mathcal{Q} \rangle|}_i$ is undefined.

The operation of filling the holes of an expansion with skeletons is defined in the obvious way, forming a new skeleton. The application of a substitution chain to a skeleton works as for types, i.e., each lifted substitution is applied in turn. The application of a lifted substitution to a skeleton is defined as follows:

(i) $\widetilde{S}(x^{\bar{\tau}}) = x^{\widetilde{S}(\bar{\tau})}$.

(ii) $\widetilde{S}(\mathcal{Q}_1 @^{\bar{\tau}} \mathcal{Q}_2) = \widetilde{S}(\mathcal{Q}_1) @^{\widetilde{S}(\bar{\tau})} \widetilde{S}(\mathcal{Q}_2)$.

(iii) $\widetilde{S}(F\mathcal{Q}) = e[\widetilde{S}(\overline{|\mathcal{Q}|}_1), \ldots, \widetilde{S}(\overline{|\mathcal{Q}|}_n)]$ where $e = S(F)$ has $n \geq 1$ holes.

(iv) $\widetilde{S}(\lambda x.\mathcal{Q}) = \lambda x.\widetilde{S}(\mathcal{Q})$.

(v) $\widetilde{S}(\lambda x^{\bar{\tau}}.\mathcal{Q}) = \lambda x^{\widetilde{S}(\bar{\tau})}.\widetilde{S}(\mathcal{Q})$.

(vi) $\widetilde{S}(\mathcal{Q}_1 \wedge \mathcal{Q}_2) = \widetilde{S}(\mathcal{Q}_1) \wedge \widetilde{S}(\mathcal{Q}_2)$.

(vii) $\widetilde{S}(\langle C, \mathcal{Q} \rangle)$ is undefined.

3.8 Type Inference Algorithms

While we informally described in section 2 the process by which one constructs a skeleton during type inference, we now make it precise.

3.8.1 Bottom-Up Constraint Collection

To define this form of inference, we first define a judgement $M \Rrightarrow \mathcal{Q}$ which means "from the term M can be constructed the initial skeleton $\mathcal{Q}$". The rules are as follows:

$$\frac{\alpha \in \mathsf{Var}_b}{x \Rrightarrow x^{\alpha}} \text{ Infer-VAR}$$

$$\frac{M \Rrightarrow \mathcal{Q}; \quad x \in \mathsf{FV}(M)}{\lambda x.M \Rrightarrow \lambda x.\mathcal{Q}} \text{ Infer-ABS-I}$$

$$\frac{M \Rrightarrow \mathcal{Q}; \quad \alpha \in \mathsf{Var}_b; \quad \alpha \notin \mathsf{var}_b(\mathcal{Q}); \quad x \notin \mathsf{FV}(M)}{\lambda x.M \Rrightarrow \lambda x^{\alpha}.\mathcal{Q}} \text{ Infer-ABS-K}$$

$$\frac{M \Rrightarrow \mathcal{Q}_1; \quad N \Rrightarrow \mathcal{Q}_2; \quad \beta, F \in \mathsf{Var}_b; \quad \mathsf{var}_b(\mathcal{Q}_1), \mathsf{var}_b(\mathcal{Q}_2), \text{ and } \{\beta, F\} \text{ are disjoint}}{MN \Rrightarrow \mathcal{Q}_1 @^{\beta} F \mathcal{Q}_2} \text{ Infer-APP}$$

The overall algorithm is then given as the following procedure:

$$\begin{aligned} \mathsf{infer}(M) = \ & \mathsf{let}\ M \Rrightarrow \mathcal{Q}, \varphi \\ & \quad \mathsf{in\ let}\ M \Rrightarrow \mathcal{Q} : \langle A, \tau \rangle / \Delta \\ & \quad\quad \mathsf{in\ let}\ C = \beta\text{-}\mathsf{unify}(\Delta) \\ & \quad\quad\quad \mathsf{in}\ C(\mathcal{Q}) \end{aligned}$$

The infer procedure is non-deterministic in the choice of names of T-variables and E-variables and also can diverge during unification.

3.8.2 Compositional Analysis with Eager Substitutions

This form of inference is slightly more complicated, because skeleton building is interleaved with β-unification and applying substitutions to skeletons. We replace the Infer-APP rule by the following inference rule:

$$\frac{\begin{array}{c} M \Rrightarrow \mathcal{Q}_1;\ N \Rrightarrow \mathcal{Q}_2;\ \beta, F \in \mathsf{Var}_b;\ \mathsf{var}_b(\mathcal{Q}_1), \mathsf{var}_b(\mathcal{Q}_2), \text{ and } \{\beta, F\} \text{ are disjoint}; \\ M \Rrightarrow \mathcal{Q}_1 : \langle A_1, \bar{\tau}_1 \rangle / \varnothing; \quad N \Rrightarrow \mathcal{Q}_2 : \langle A_2, \bar{\tau}_2 \rangle / \varnothing; \quad C = \beta\text{-}\mathsf{unify}(\{\bar{\tau}_1 \doteq \bar{\tau}_2 \to \beta\}) \end{array}}{MN \Rrightarrow C(\mathcal{Q}_1 @^{\beta} F \mathcal{Q}_2)} \text{ Infer-APP-Eager}$$

The overall algorithm is then given as the following procedure:

$$\mathsf{infer}(M) = \mathcal{Q} \quad \text{where } M \Rrightarrow \mathcal{Q}$$

3.8.3 Compositional and Incremental Analysis with Lazy Substitutions

This form of inference is a slight variation on the previous one, which differs only by constructing a skeleton with suspended substitutions instead of applying the substitutions to the skeleton. The new Infer-APP-Lazy rule is used instead of the Infer-APP or Infer-APP-Eager rules. Infer-APP-Lazy is the same as Infer-APP-Eager, except that instead of applying the substitution as in $C(\mathcal{Q}_1 @^\beta F \mathcal{Q}_2)$, it constructs a skeleton with a suspended substitution as in $\langle C, \mathcal{Q}_1 @^\beta F \mathcal{Q}_2 \rangle$. The same definition of infer is reused.

3.9 Finite Ranks

Up until now we have ignored the fact that in general β-unification is non-terminating. In particular, λ-terms that are not strongly normalizable generate constraint sets that cause any algorithm for β-unification to run forever. So in practice we set a bound on how long we allow β-unification to proceed by restricting the maximum "rank" which a type may possess in a derivation.

Informally, the rank of a type τ is a measure on how deep "$\wedge$" occurs in τ; more precisely, it counts the maximum number of times (plus one) which a path from the root of τ visits the left of a "$\rightarrow$" to reach an occurrence of "$\wedge$". A formal definition is by induction in types:

(i) $\mathsf{Rnk}(\alpha) = 0$.

(ii) $\mathsf{Rnk}(\tau \rightarrow \bar{\tau}) = \begin{cases} 0 & \text{if } \mathsf{Rnk}(\tau) = \mathsf{Rnk}(\bar{\tau}) = 0, \\ \max\{1 + \mathsf{Rnk}(\tau), \mathsf{Rnk}(\bar{\tau})\} & \text{otherwise.} \end{cases}$

(iii) $\mathsf{Rnk}(\tau_1 \wedge \tau_2) = \begin{cases} 1 & \text{if } \mathsf{Rnk}(\tau_1) = \mathsf{Rnk}(\tau_2) = 0, \\ \max\{\mathsf{Rnk}(\tau_1), \mathsf{Rnk}(\tau_2)\} & \text{otherwise.} \end{cases}$

(iv) $\mathsf{Rnk}(F\,\tau) = \mathsf{Rnk}(\tau)$.

Given a set Δ of n constraints $\{\tau_1 \doteq \tau_2, \ldots, \tau_{2n-1} \doteq \tau_{2n}\}$, we define

$$\mathsf{Rnk}(\Delta) = \max\{\mathsf{Rnk}(\tau_1), \ldots, \mathsf{Rnk}(\tau_{2n})\}.$$

This is a straightforward easy-to-implement definition of $\mathsf{Rnk}(\ )$. However, the test to forcibly terminate β-unification, once a given maximum rank K is exceeded, is not to test whether $\mathsf{Rnk}(\Delta) \geq K$ after every step of the algorithm.[⋆⋆⋆⋆] Rather, if Δ_0 is the initial constraint set and C is the chain of small substitutions constructed after $n \geq 1$ rewrite steps by the algorithm, it is necessary to test whether $\mathsf{Rnk}(C(\Delta_0)) \geq K$. Call $\mathsf{Rnk}(C(\Delta_0))$ the *global rank* of the initial constraint set Δ_0 after n rewrite steps, which is non-decreasing as a function of n.

There are different ways of calculating the global rank. One way is proposed in [11], which is good enough for proving the theorems in that report,

[⋆⋆⋆⋆] In fact, there are rewriting strategies for the algorithm of figure 1 such that $\mathsf{Rnk}(\Delta)$ never exceeds 3.

but which is also cumbersome to implement. An alternate way of calculating the global rank is to keep markers for the "order" of types occurring in constraints and to keep a minimum-rank counter for occurrences of $\wedge$ that have been discarded by simplification of the constraint set. This is explained next.

3.10 Keeping Track of The Global Rank

In order to keep track of the global rank, we extend types with markers for the *order* of positions in the types and we pair each constraint set with a *minimum rank*. Keeping track of these values is necessary because of the way the $\mathsf{simplify}$ function breaks apart constraints with matching outermost type constructors and discards solved constraints.

We implement *order-marked* types by using an additional unary type constructor ι which causes its type argument to be viewed as occurring at a higher order. We forbid ι from occurring inside the type arguments of $\wedge$ and $\rightarrow$, because we do not need this. In the following presentation, we will allow the metavariable F to range over uses of ι in addition to expansion variables. A *constraint-with-order* is a pair of two types $\boldsymbol{F}\tau_1$ and $\boldsymbol{F}\tau_2$, written $\boldsymbol{F}\tau_1 \doteq \boldsymbol{F}\tau_2$, where τ_1 and τ_2 do not mention ι. Let $\mathsf{order}(F_1 \cdots F_n)$ count the number of items in the sequence $F_1, \ldots, F_n$ that are ι. Let a constraint set *with orders and minimum rank* be a set Δ of constraints-with-order paired with a minimum rank k (a natural number), written (k, Δ). The function init is now defined to convert a constraint set into a constraint set with orders and minimum rank. Let $\mathsf{init}(\Delta) = (0, \Delta)$. The operations of substitution and expansion variable application are extended to constraint sets with orders and minimum rank by component-wise distribution to the types inside the constraints. The $\mathsf{simplify}$ function gets a new definition as follows:

$$
\begin{aligned}
&\mathsf{simplify}(k, \{\boldsymbol{F}(\tau_1 \rightarrow \tau_2) \doteq \boldsymbol{F}(\tau_1' \rightarrow \tau_2')\} \cup \Delta) \\
&\qquad = \mathsf{simplify}(k, \{\boldsymbol{F}\iota\tau_1' \doteq \boldsymbol{F}\iota\tau_1, \boldsymbol{F}\tau_2 \doteq \boldsymbol{F}\tau_2'\} \cup \Delta), \\
&\mathsf{simplify}(k, \{\boldsymbol{F}(\tau_1 \wedge \tau_2) \doteq \boldsymbol{F}(\tau_1' \wedge \tau_2')\} \cup \Delta) \\
&\qquad = \mathsf{simplify}(\max(k, \mathsf{order}(\boldsymbol{F}) + 1), \{\boldsymbol{F}\tau_1 \doteq \boldsymbol{F}\tau_1', \boldsymbol{F}\tau_2 \doteq \boldsymbol{F}\tau_2'\} \cup \Delta), \\
&\mathsf{simplify}(k, \Delta) \\
&\qquad = (k, \Delta) \text{ otherwise.}
\end{aligned}
$$

Notice that solved constraints are no longer discarded. Solved constraints must be kept because the types in a solved constraint will contain normal type variables and possibly also expansion variables, and substitutions generated later for these variables may result in occurrences of $\wedge$ being inserted at higher-rank positions. In an implementation, solved constraints should be marked so that they can be efficiently skipped over by the part of the unification algorithm that picks the constraint to reduce.

The rest of the β-unification algorithm definitions in figure 1 are lifted to constraint sets with orders and minimum rank in the obvious straightforward way.

Finally, the definition of success needs some changes. The *rank* of a constraint-with-order $(\boldsymbol{F}\tau \doteq \boldsymbol{F}\tau')$ where both τ and τ' are ι-free, written $\mathsf{Rnk}(\boldsymbol{F}\tau \doteq \boldsymbol{F}\tau')$, is 0 if $\mathsf{Rnk}(\tau) = \mathsf{Rnk}(\tau') = 0$ and otherwise is $\mathsf{order}(\boldsymbol{F}) + \max(\mathsf{Rnk}(\tau), \mathsf{Rnk}(\tau'))$. The *rank* of a constraint set with orders and minimum rank (k, Δ) is given by $\mathsf{Rnk}(k, \Delta) = \max\bigl(\{k\} \cup \{\, \mathsf{Rnk}(\tau \doteq \tau') \mid (\tau \doteq \tau') \in \Delta \,\}\bigr)$. The definition of success for the rank-k restriction of β-unification is as follows. An instance Δ of β-unification *succeeds at rank k* iff there is a sequence of $n+1$ rewrite steps such that

$$\mathsf{init}(\Delta) \underset{\mathsf{init}}{\Longrightarrow} (k_0, \Delta_0) \overset{S_1}{\underset{r_1}{\Longrightarrow}} (k_1, \Delta_1) \overset{S_2}{\underset{r_2}{\Longrightarrow}} \cdots \overset{S_n}{\underset{r_n}{\Longrightarrow}} (k_n, \Delta_n),$$

such that $\tau = \tau'$ for every constraint $(\tau \doteq \tau') \in \Delta_n$, and such that $\mathsf{Rnk}(k_n, \Delta_n) \leq k$.

Because $(k, \Delta) \overset{S}{\underset{r}{\Longrightarrow}} (k', \Delta')$ implies $\mathsf{Rnk}(k', \Delta') \geq \mathsf{Rnk}(k, \Delta)$, the rank-$k$ β-unification algorithm can stop and report failure whenever it reaches a state (k', Δ) such that $\mathsf{Rnk}(k', \Delta) > k$. It is more difficult to show that the algorithm can only iterate for a bounded number of steps before the rank increases or all constraints become solved; see [11] for some information about this for another definition of β-unification.

4 An Aside: Using XML Technologies in Type-Based Analysis

Our implementations of System $\mathbb{I}$ have made heavy use of XML (the Extensible Mark-up Language [3]) as a framework for manipulating and communicating structured data. The input (currently just λ-terms and option settings) to and all of the output (skeletons, types, constraint sets, substitutions, etc.) from our analysis implementations are represented as XML.

The XML standard is far from ideal in many respects, and offers insignificant technical advantages over the S-expression technology which has existed for decades [13]. In many respects, it suffers from being a descendant of SGML [1], which has led to the inclusion of many features interfering with extensibility and many arbitrary restrictions. Despite these shortcomings, XML does have the advantage of being the first structured data format that academia and industry are willing to agree upon. Having this consensus allows us to finally move the *lingua franca* of data storage and communication beyond bit vectors.

XML is highly promising for those working in programming languages and program analysis as well as many other closely aligned areas. One potential benefit is that it can provide a way to standardize on a concrete "universal" abstract syntax for many languages. Having a standardized encoding of the abstract syntax of numerous languages within XML would allow for the development of tools and analysis techniques that could be applied independently of the actual languages used.

However, there are still many problems with XML that must be overcome which we have encountered in our work. One problem is the difficulty in representing types compactly. This is actually two subproblems. The first subproblem is that there is no standard way of representing DAGs (directed acyclic graphs) with sharing in XML. This is a problem because often the sizes of types become exponentially larger when expressed as trees rather than as DAGs. Although we could devise our own way of representing DAGs within XML (encoding DAGs as trees), this interferes with our goal of convenient use of standard XML tools such as XSLT processors, so we have not done this. We may end up doing this, but we are hoping someone else will standardize a solution for this first and adapt technologies like XSLT. The second subproblem is that the present way the XML standard encodes trees as bit vectors is extremely space inefficient. There is already work on multiple standards for improving the efficiency of XML at representing trees, but no standard has been accepted and none is widely implemented. The combined effect of these subproblems is that for certain terms our System $\mathbb{I}$ analysis engines can successfully infer a principal typing, but will be unable to construct the XML output because it would exceed the available memory.

Another problem is the lack of a reasonable standard for imposing types on the structure of data represented in XML. When XML was originally proposed, Document Type Definitions (DTDs) (another legacy of SGML) were the recommended mechanism for describing document structure. DTDs are problematic because they do not offer a very rich language and are difficult to manipulate as they not stored as XML documents themselves. Recently the W3C XML Schema [6] language was developed, but it is extremely complex and lacks useful specification and extensibility features such as parametric polymorphism. Other competing standards exist, like Relax NG [4], but they are not yet widely implemented or accepted and we have not yet had time to evaluate them for our purposes. What this means for us is that currently the types we use to constrain our XML data are overly liberal and permit many possibilities that we would like them to exclude.

Finally, support for manipulating XML documents within common programming languages is inadequate. In particular, for our purposes there is effectively *no* XML support available for Standard ML, so we have had to "roll our own" for the one implementation we did in SML. Some languages (e.g., Java) do have reasonable libraries for working with XML documents, but we have found they are still cumbersome to use. Research into extending languages with first class facilities for more easily manipulating XML is ongoing [7,15] but such facilities are still far from commonplace.

5 Future Directions

While the promise of System $\mathbb{I}$ is great, there still remains a significant amount of work to be done towards allowing existing languages to benefit from this kind of compositional analysis. It is particularly important that the analysis be extended beyond the pure λ-calculus to support common language features. Presently Washburn and Wells are investigating a new, unpublished extension to System $\mathbb{I}$ which adds pattern matching, tuples, and unit values. Research still needs to be done on integrating recursive definitions and imperative features (e.g., assignments, exceptions, input/output). Primitive support for recursion must be added because the Y combinator is untypable in System $\mathbb{I}$ (because it is not strongly normalizing (SN) for β-reduction).

Additionally, because the intersection type constructor is not idempotent in System $\mathbb{I}$ and because the typing rules do not allow sharing of assumptions between multiple premises, a System $\mathbb{I}$ typing derivation for a λ-term in effect encodes an *exact* analysis of the term. This analysis is exact in the sense that the principal typing obtained contains information sufficient to answer *every* possible question about the observable behavior of the term. The finite-rank restriction of System $\mathbb{I}$ merely decides when to give up on finding an analysis, and does not affect the precision of the analysis when one is found. For practical use, System $\mathbb{I}$ needs to be extended with the ability to represent cruder analyses, because the exact analysis is far too expensive in both time and space. One possible approach would be to make the intersection type constructor associative, commutative, and idempotent (ACI) beyond rank k when used with the rank-k restriction. We are currently exploring the issues involved in this.

There is presently ongoing research into attempting to merge the strengths of System $\mathbb{I}$, the branching type system of Wells and Haack [18], and the system of Amtoft and Turbak[2] and its support for tagged intersection and union types as well as subtyping. This could allow for principal typing derivations with less redundancy and could make it easier to implement local transformations on terms while preserving the correctness of the derivations. Also, as mentioned previously there is also active research into a version of β-unification that does not require renaming. An overriding goal in research directions will be to try to achieve greater simplicity in design and presentation than System $\mathbb{I}$.

References

[1] American National Standards Institute and International Organization for Standardization. *Information processing: text and office systems: Standard Generalized Markup Language (SGML).* American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1985.

[2] Torben Amtoft and Franklyn Turbak. Faithful translations between polyvariant

flows and polymorphic types. In *Programming Languages & Systems, 9th European Symp. Programming*, volume 1782 of *LNCS*, pages 26–40. Springer-Verlag, 2000.

[3] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible Markup Language (XML) 1.0 (second edition). W3C Recommendation - `http://www.w3.org/TR/2000/REC-xml-20001006`, October 2001.

[4] James Clark and Murata Makoto. RELAX NG Specification. Oasis Committee Specification - `http://www.oasis-open.org/committees/relax-ng/spec-20011203.html`, December 2001.

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[6] David C. Fallside. XML Schema Part 0: Primer. W3C Recommendation - `http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/`, May 2001.

[7] Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language (preliminary report). In *WebDB (Informal Proceedings)*, pages 111–116, 2000.

[8] Trevor Jim. What are principal typings and what are they good for? Tech. memo. MIT/LCS/TM-532, MIT, 1995.

[9] Assaf J. Kfoury. Beta-reduction as unification. In D. Niwinski, editor, *Logic, Algebra, and Computer Science (H. Rasiowa Memorial Conference, December 1996), Banach Center Publication, Volume 46*, pages 137–158. Springer-Verlag, 1999.

[10] Assaf J. Kfoury, Harry G. Mairson, Franklyn A. Turbak, and J. B. Wells. Relating typability and expressibility in finite-rank intersection type systems. In *Proc. 1999 Int'l Conf. Functional Programming*, pages 90–101. ACM Press, 1999.

[11] Assaf J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *Conf. Rec. POPL '99: 26th ACM Symp. Princ. of Prog. Langs.*, pages 161–174, 1999. Superseded by [12].

[12] Assaf J. Kfoury and J. B. Wells. Principality and type inference for intersection types using expansion variables. Supersedes [11], August 2002.

[13] John L. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[14] Robin Milner. A theory of type polymorphism in programming. *J. Comput. System Sci.*, 17:348–375, 1978.

[15] Santiago M. Pericas-Geertsen. *XML-Fluent Mobile Ambients*. PhD thesis, Boston University, 2001.

[16] Geoffrey Washburn, Bennett Yates, Bradley Alan, J. B. Wells, and Assaf Kfoury. A tool for experimenting with system $\mathbb{I}$. `http://types.bu.edu/modular/compositional/experimentation-tool/`.

[17] J. B. Wells. The essence of principal typings. In *Proc. 29th Int'l Coll. Automata, Languages, and Programming*, volume 2380 of *LNCS*, pages 913–925. Springer-Verlag, 2002.

[18] J. B. Wells and Christian Haack. Branching types. In *Programming Languages & Systems, 11th European Symp. Programming*, volume 2305 of *LNCS*, pages 115–132. Springer-Verlag, 2002.

[19] Bennett Yates. Intersection types with expansion variables: The case of associative and commutative $\wedge$ with a new formulation of substitution. Unpublished.