LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Incremental Polymorphism

Computation Structures Group Memo 329
June 6, 1991

Shail Aditya

Rishiyur S. Nikhil

To appear in Proc. Functional Programming Languages and Computer Architecture, Cambridge, MA, Aug 28-30, 1991

This paper describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-89-J-1988.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Incremental Polymorphism

Shail Aditya Rishiyur S. Nikhil

June 6, 1991

Abstract

The Hindley/Milner polymorphic type system has been adopted in many programming languages because it provides the convenience of programming languages like Lisp along with the correctness guarantees that come with static type-checking. However, programming environments for such languages are still not as flexible as those for Lisp. In particular, the style of incremental, top-down program development possible in Lisp is precluded because the type inference system is usually formulated as a "batch system" that must examine definitions before their uses. This may require large parts of the program to be recompiled when a small editing change is performed.

In this paper, we attempt to strike a balance between the apparently conflicting goals of incremental, top-down programming flexibility and static type-checking. We present an incremental typing mechanism in which top-level phrases can be compiled one by one, in any order, and repeatedly (due to editing). We show that the incremental type system is sound and complete with respect to the more traditional "batch system". The system derives flexibility from the inherent polymorphism of the Hindley/Milner type system and minimizes the overhead of book-keeping and recompilation. Our system is implemented and has been in use by dozens of users for more than two years.

1 Introduction

Modern computing environments strive for several desirable features: the environments should support the development of reliable programs, *i.e.*, they should be able to detect as many programming errors as early as possible; the environments should be robust, *i.e.*, they must gracefully report all errors and exceptions as and when they occur; finally, the environments should have flexible and interactive facilities for editing, testing and debugging of programs.

Strongly typed languages meet the first goal by guaranteeing that "type-consistent" programs will not incur run-time type-errors. Recent programming languages based on the Hindley/Milner type system [4, 8] also provide the convenience of type polymorphism and automatic type inference. But most programming environments for such languages have to compromise the flexibility of incremental, top-down program development in order to achieve this "type-consistency" over the whole program. This is because the Hindley/Milner type system is usually formulated as a "batch system" that must examine the definitions before they can be used and the complete program before any of its parts can be exercised. The problem is further complicated due to polymorphism where the meaning of "type-consistency" is one of inclusion, rather than equality.

In this paper, we will address the issue of providing a robust and interactive programming environment for Id, which is a polymorphic, strongly typed, incrementally compiled, parallel programming language

Current address for both authors: Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, Massachusetts 02139. Current Internet e-mail address: `shail@abp.lcs.mit.edu` and `nikhil@abp.lcs.mit.edu`.
This paper describes research done at the Laboratory for Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-89-J-1988.

developed at the Laboratory for Computer Science, MIT [11]. Id's typing mechanism is based on the Hindley/Milner type inference system, but it differs from its traditional description in that it is "incremental" in nature. We have modified and extended the standard Hindley/Milner type inference algorithm to facilitate incremental development and testing of programs. In this paper, we will describe this incremental algorithm used in Id. We will also show its correctness (soundness and completeness) with respect to the standard formulation of the Hindley/Milner type inference algorithm that examines the complete program. Our goal is to produce the same typings as obtained *via* the standard algorithm, only that we produce them incrementally. For convenience, we will refer to the traditional type inference mechanism as the "batch system" and our mechanism as the "incremental system" throughout the paper.

The paper is organized as follows. Section 2 discusses a few other interactive programming environments and motivates the incremental approach taken in Id. Section 3 establishes the notations used in this paper. Section 4 describes the issues in incremental type inference under an interactive environment *via* several examples. Section 5 describes our incremental type inference system in detail. In Section 6 we show its correctness. Section 7 extends the incremental algorithm to handle complex editing situations. In Section 8 we discuss the complexity of our system and briefly describe some possible optimizations that are detailed in appendix A. Finally, in Section 9 we summarize our results and compare them with the related work in this field.

2 Background

Nikhil in [10] pointed out the disparity between the goals of an incremental programming environment, and ML-like type inference system. ML [2, 9] is interactive, but a session in ML is essentially a large lexically nested ML program. Each toplevel definition has the rest of the session as its scope. Thus, editing an earlier definition may force the user to recompile and reload all the intermediate definitions that used it[1]. In building large systems, the recompilation and reloading of large pieces of potentially unrelated code, just to recreate the same environment every time a small error is detected, is at best, quite annoying.

In Miranda [16, 17], this problem is resolved by making the unit of compilation to be a whole file (also called a "Miranda script"). Definitions within a file may appear in any order and the compiler is responsible for reordering them during compilation. The interactive session only evaluates expressions using definitions from the current script. But, definition level incrementality has been lost, and editing forces entire files to be recompiled.

In contrast, Lisp programming environments smoothly integrate the editor, the compiler, and the read-evaluate-print loop. The unit of compilation in these systems is a single top-level definition. They allow the user to furnish multiple top-level definitions, either together, or one by one in any order, resolving global references to other definitions automatically by dynamic linking. The user can test, debug, and edit these definitions incrementally, without waiting to write the complete program or having to recompile a substantial fraction of the definitions already supplied.

In Id, we attempt to achieve the flexibility of Lisp-like environments along with static type-checking. The programming environment for Id, called "Id World"[12], smoothly integrates the editor, the Id compiler, and the underlying execution vehicle. Like Lisp, our unit of compilation is a single top-level definition. Each compiled definition is accumulated into a flat global environment, implying that edited definitions are immediately and automatically made available to other definitions that use them. Simple interprocedural book-keeping maintains type-consistency between the definition and the uses of each top-level identifier. The book-keeping mechanism derives flexibility from the polymorphism of the definitions, flagging only inconsistent definitions for recompilation. As a consequence, we permit out of order compilation, redefinition, and editing with minimum overhead, while still guaranteeing type correctness before program execution. We also allow executing partially defined Id programs, where the undefined identifiers simply generate an exception if actually used at run-time. Again, the incremental book-keeping mechanism helps in incorporating the missing parts as and when they become available with minimum overhead.

[1] SML's module mechanism gives some relief in this respect, due to separate compilation.

3 Syntax and Notation

In this section, we present a brief overview of the notation used in this paper. Readers are referred to [5, 14] for details. Those already familiar with the Hindley/Milner type system may just skim this section for the notation — there are no new concepts introduced here.

3.1 The Expression Mini-Language

The basic expression language used in describing the Hindley/Milner type system is fairly small. We use the same formulation as in [4, 14] with only slight modifications to suit our incremental system. The syntax of the mini-language appears below:

$$
\begin{array}{rcll}
x, y, z \in \text{Identifiers} & & & (1)\\
c \in \text{Constants} & ::= & \{\texttt{true}, \texttt{false}, 1, 2, \ldots\} & (2)\\
e \in \text{Expressions} & ::= & \quad c & (3)\\
 & & \mid \; x & \\
 & & \mid \; \lambda x.\, e_1 & \\
 & & \mid \; e_1 \; e_2 & \\
 & & \mid \; \texttt{let} \; x = e_1 \; \texttt{in} \; e_2 & \\
B \in \text{Bindings} & ::= & x = e & (4)\\
P \in \text{Programs} & ::= & B_1;\; B_2; \cdots;\; B_n;\; \texttt{it} = e & (5)
\end{array}
$$

The above mini-language retains the notion of top-level, independent bindings as units of compilation. A complete program is a set of such bindings. The final binding is special[2] in that it represents a program query and is also evaluated after being compiled within the current environment.

We will freely (and informally) use tuple expressions and tuple bindings in this mini-language because tuples can always be simulated by the function type constructor ($\rightarrow$) alone.

3.2 The Type Language

The standard Hindley/Milner type language is inductively defined as follows:

$$
\begin{array}{rcll}
\pi \in \text{Type-Constructors} & = & \{\mathit{int}, \mathit{bool}, \ldots\} & (6)\\
\alpha,\beta \in \text{Type-Variables} & = & \{\texttt{*0}, \texttt{*1}, \ldots\} & (7)\\
\tau \in \text{Types} & ::= & \quad \pi & (8)\\
 & & \mid \; \alpha & \\
 & & \mid \; \tau_1 \rightarrow \tau_2 & \\
\sigma \in \text{Type-Schemes} & ::= & \quad \tau & (9)\\
 & & \mid \; \forall \alpha.\, \sigma_1 &
\end{array}
$$

3.3 Type Notation

A **Type Environment** maps identifiers to type-schemes. All type variables of a type τ are considered to be **free** in that type. The quantified type-variables of a type-scheme are taken to be **bound** in that type-scheme and the other type-variables are taken to be **free**. This definition extends pointwise to type environments. We will denote the free type-variables of T by $\mathit{tyvars}(T)$, where T could either be a type, a type-scheme, or a type environment.

[2] This is adapted from the ML interactive runtime environment where the last expression evaluated is referred to as "`it`".

A **Substitution** S maps type-variables to types. By extension, we can apply substitutions to types, type-schemes, or type environments, in each case only operating on their free type-variables. Given a type-scheme $\sigma = \forall\alpha_1 \cdots \alpha_n.\, \tau$, an **Instantiation** $\sigma \succ \tau'$ is defined by a substitution S for the bound variables of σ so that $S\tau = \tau'$. The instantiation $\sigma_1 \succeq \sigma_2$ is valid if $\sigma_1 \succ \tau_2$ (where $\sigma_2 = \forall\beta_1 \cdots \beta_m.\, \tau_2$) and no β_j is free in σ_1. The key point to remember is that substitutions affect only free type-variables, while instantiations operate only on bound type-variables.

Given a type τ and a type environment TE, we define $close(TE, \tau) = \forall\alpha_1 \cdots \alpha_n.\, \tau$ where $\{\alpha_1, \ldots, \alpha_n\} = tyvars(\tau) - tyvars(TE)$. When $tyvars(TE) = \phi$, we may also write simply $close(\tau)$ instead of $close(TE, \tau)$.

4 Issues in Incremental Typing

In this section, we will describe the various issues that need to be addressed during incremental type inference by means of simple examples[3].

4.1 Forward References

In Id, all top-level definitions are compiled into a single global environment. So, it is possible and convenient to use functions that are defined later, as the following example shows[4]:

```
def f x = (x+1):(g x);                 % int -> (list int)

def g x = x:nil;                       % *0 -> (list *0)
```

We have shown the inferred type for each of the definitions as a comment appearing to the right of the definition[5]. Assuming that the above definitions are compiled in the order of their appearance, the function `f` uses `g` inside its body as a function with type `(int -> (list int))` without actually knowing anything about it.

When `g` is compiled, `f` "sees" its definition and the system should make sure that all its previous uses are "consistent" with its definition. In later sections, we will describe in detail how this consistency check is formulated and verified. For now, it suffices to say that use of an identifier should have a type that is an instance of the type inferred at its definition. In the above example, it is indeed the case, the type of `g`, `(int -> (list int))`, used within `f` is an instance of the defined type of `g`, `(*0 -> (list *0))`[6]. Therefore, `f` need not be recompiled even though it used `g` before it was defined.

4.2 Editing

Going a step further, we may edit `g` as follows:

```
def g x = x:(x-1):nil;                 % int -> (list int)
```

This restricts the type of `g`, but its use inside `f` is still valid, and no recompilation is necessary. On the other hand, if we had redefined `g` as,

```
def g x = x-1;                         % int -> int
```

then the inferred type of `g` no longer matches its use inside `f` and the system should detect it and report an error. Note that this analysis is independent of the order of the original definition of `f` and `g`, or for that matter, any other definitions that appear temporally in between the definitions of `f` and `g`. Such independent definitions are never disturbed; all recompilation requirements, if any, apply only to the definitions that fail the consistency check.

[3]Even though all our analysis is based on the mini-language given in section 3, our examples use the full Id syntax for convenience and clarity. In [5], we show a simple translation from Id to this mini-language.

[4]In Id, toplevel function definitions are introduced with the keyword `def` and terminated by a semi-colon (`;`). Also, colon (`:`) is the infix cons operator. Text following a percent (`%`) is taken to be a comment and is ignored.

[5]The type variables appearing in a type are assumed to be implicitly universally quantified at the outermost unless otherwise stated or clear by the context.

[6]This instantiation uses the simple substitution $S = \{*0 \mapsto \texttt{int}\}$ for the bound type-variable `*0`.

4.3 Mutual Recursion

The situation becomes more complicated with mutually recursive functions, which have to be type-checked together in the Hindley/Milner type system. In the incremental system, such definitions may be compiled separately. We have to either rule out such cases by requiring that mutually recursive definitions be supplied together, or incrementally detect definitions that become mutually recursive and handle them appropriately. Simple book-keeping, as described in section 4.1, may fail to catch type-errors embedded across mutually recursive definitions, as the following example shows:

```
def f x = g f;                    % *0 -> *1
...
def g x = f g;                    % *2 -> *3
```

The above two definitions are mutually recursive and are rejected by the batch system[7]. The incremental system infers the type of `f` to be `(*0 -> *1)`, as shown above, assuming the type of `g` to be `(*0 -> *1) -> *1)`. Similarly, when `g` is compiled, its type is obtained as `(*2 -> *3)`, assuming the type of `f` to be `((*2 -> *3) -> *3)`. Note that in both cases, the assumed types are instances of their corresponding inferred types, and the simple consistency checking used in section 4.1 is not sufficient to catch the type-error in the above program.

The following example shows that without explicit book-keeping of mutually recursive definitions, the incremental system may, in fact, compute unsound types even when there is no overall type-error.

```
def K x y = x;                    % *0 -> *1 -> *0

def f x = (g f) x;                % *2 -> *3

def g x = K x (x:(f x));          % *4 -> *4
```

The `K` function simply returns its first argument. The inferred types of `f` and `g` individually are as shown. The uses of both `f` and `g` are instances of their inferred types, so no consistency error is present. But when supplied together and taking into account that `f` and `g` are mutually recursive, the correctly computed Hindley/Milner type of `g` should be `((*4 -> (list *4)) -> (*4 -> (list *4)))` and that of `f` should be `(*4 -> (list *4))`.

4.4 Editing and Type Relaxation

It is possible during editing that the type of a definition gets relaxed and therefore must be reflected in other definitions that use it. Consider the following example:

```
def f x y = (x+1);                % int -> *0 -> int

def g x y = f x (y+1);            % int -> int -> int
```

The type of `f` constrains the type of `g`'s argument `x` to be `int`. Now, if we decide to edit the function `f` so as to relax its type,

```
def f x y = x;                    % *1 -> *0 -> *1
```

the type of `g` that was earlier constrained to be `(int -> int -> int)` can now also be relaxed to `(*2 -> int -> *2)`. The system should be able to detect this as well. Note that this relaxation does not create unsound types but may render the original types as incomplete or non-principal.

Constraint relaxation may also occur due to a change in mutual recursion among definitions. Consider the following example:

[7] The Hindley/Milner type system does not handle infinite types and flags them as type-errors. In this example, the type of both `f` and `g` are infinite.

```
def fst (x,y) = x;                    % (*0,*1) -> *0

def h x = if true then
            x
          else fst (t x x);           % *2 -> *2

def t x y = h x,h y;                  % *2 -> *2 -> (*2,*2)
```

Since `h` and `t` are mutually recursive, the type of the two arguments of `t` are constrained to be the same. Now, if we edit the definition for `h` to be the simple identity function,

```
def h x = x;                          % *2 -> *2
```

then the constraint on the arguments of `t` is no longer present since `h` can now be instantiated differently. Therefore, the type of `t` can be relaxed to `(*3 -> *4 -> (*3,*4))`. The system should be able to detect this and flag the recompilation of `t`.

5 The Incremental Type Inference System

Our incremental system is based on the standard Hindley/Milner inference rules given in the literature [4, 3, 14], so we will not describe those inference rules again. We follow the description of [14] which expresses the rules for Instantiation and Generalization implicitly. This has the advantage of making the inference rules completely deterministic[8], and we always infer a type for an expression instead of a type-scheme[9].

In this section, we will describe our incremental book-keeping strategy and the incremental type inference algorithm that uses it.

5.1 Incremental Book-Keeping

The basic idea in incremental compilation is to be able to compute some desired compile-time properties for an aggregate of identifiers in an incremental fashion. This aggregate forms the **identifier namespace** that we operate in. For our purposes, this is the set of all top-level identifiers. Our first step is to define a "property".

Definition 1 *A* **property** $\mathcal{P} = (\mathcal{D}, \sqsubseteq)$ *is characterized by a domain of values* $\mathcal{D}$ *partially ordered*[10] *by the relation* $\sqsubseteq$*. Given two values,* $v_1, v_2 \in \mathcal{D}$ *for a property* $\mathcal{P} = (\mathcal{D}, \sqsubseteq)$*, we say that* v_1 *is consistent with* v_2 *if and only if* $v_1 \sqsubseteq v_2$*.*

The domain of values is simply a syntactic set of values with some structural relationship defined among its elements. The domain must also contains a special element "$\perp$" (read "bottom") that corresponds to the default property value assigned to as yet undefined identifiers in the namespace.

The interdependences among the properties of identifiers at various times during incremental compilation is maintained *via* sets of "assumptions" defined below.

Definition 2 *An* **assumption** $(x, y, v_y) \in \mathcal{A}$ *is a triple consisting of an* **assumer** x*, an* **assumee** y*, and an* **assumed-value** $v_y \in \mathcal{D}$*, for the assumee's property* $\mathcal{P} = (\mathcal{D}, \sqsubseteq)$*.* $\mathcal{A}$ *is termed as the* **assumption domain***. An* **assumption set** A_x *for the assumer* x *is a set of all such assumptions made by* x *and can be written as a map from the assumees to their assumed-values.*

Each assumption domain has an associated consistency checking function. An **assumption check** C *is a predicate that verifies the assumed-value* v_y *of an assumee from a given assumption set against its current value* v'_y *available in the environment for consistency. This check may use the property predicate* $\sqsubseteq$ *for this purpose.*

[8] This means that exactly one rule will apply to a given expression.
[9] The equivalence of the rules appearing in [4] and those in [14] has been shown in [3].
[10] A partial order on a domain is a reflexive, transitive, and anti-symmetric binary relation on the elements of the domain.

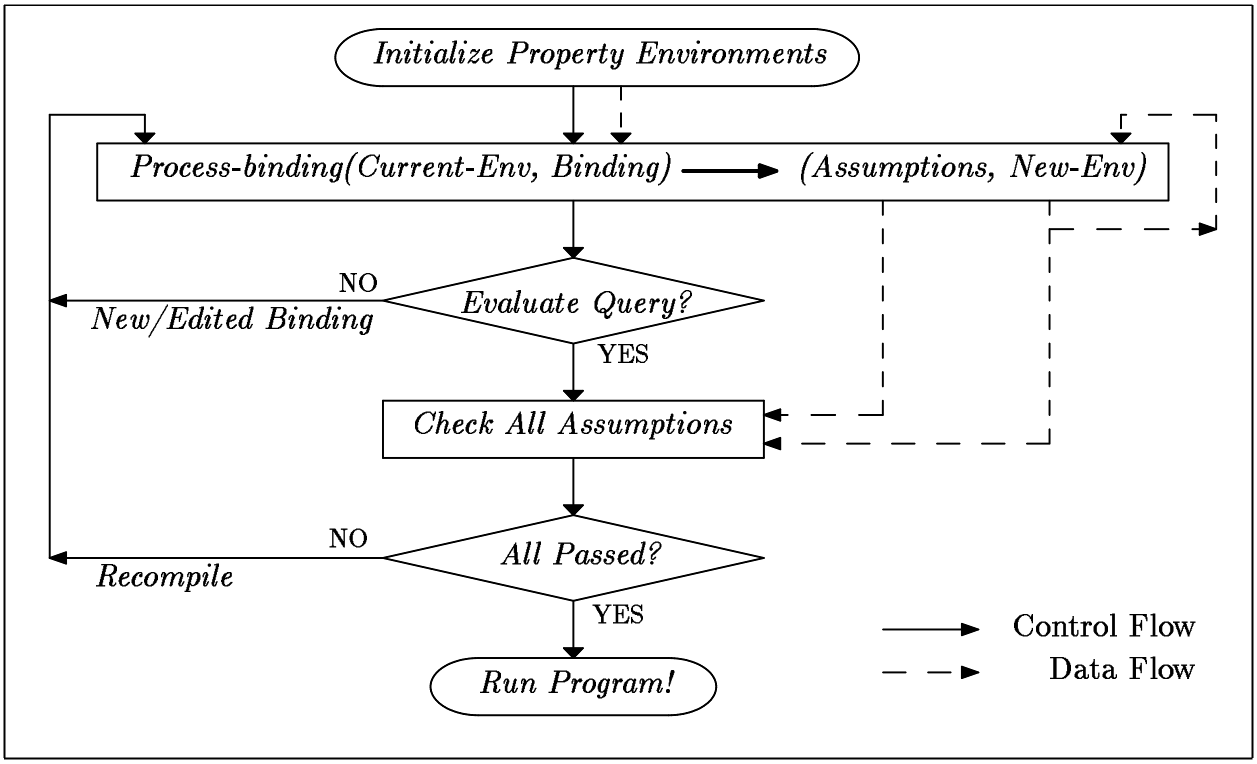


Figure 1: Overall Plan for Incremental Property Maintenance.

Assumption domains are characterized by the properties they record and the assumption checks they employ in order to verify consistency. Several assumption domains may be associated with the same property that use different assumption checks. We will see examples of this later on. Also note that an assumption set for an assumer may contain several assumptions for the same assumee corresponding to its various occurrences in the definition of the assumer.

The union of all the property mappings of namespace identifiers to their property values constitutes a **compilation environment**. The sets of assumptions associated with each assumer identifier make up the book-keeping overhead of the compilation environment.

5.2 Overall Plan for Incremental Analysis

The overall scheme for incremental property computation and maintenance appears in Figure 1. Essentially, we process each top-level binding individually, accumulating its assumptions and property values in the current environment. If we need to evaluate a query, we first check whether all the accumulated assumptions are consistent with respect to the latest environment, and proceed to evaluate the query only if all the assumptions pass their checks. Otherwise, the failing definitions are flagged for recompilation.

The success or failure of the above mechanism in computing the desired properties correctly and efficiently depends upon several factors. These factors include, the semantic characteristics of the accumulated property domains, their interdependences and computation algorithms, the various kinds of assumptions collected and their respective assumption checks, and the incremental compilation environment history and its maintainance. Now, we will use this incremental property maintenance strategy to compute the types of top-level definitions within the well defined framework of Hindley/Milner type inference system.

5.3 Properties and Assumptions

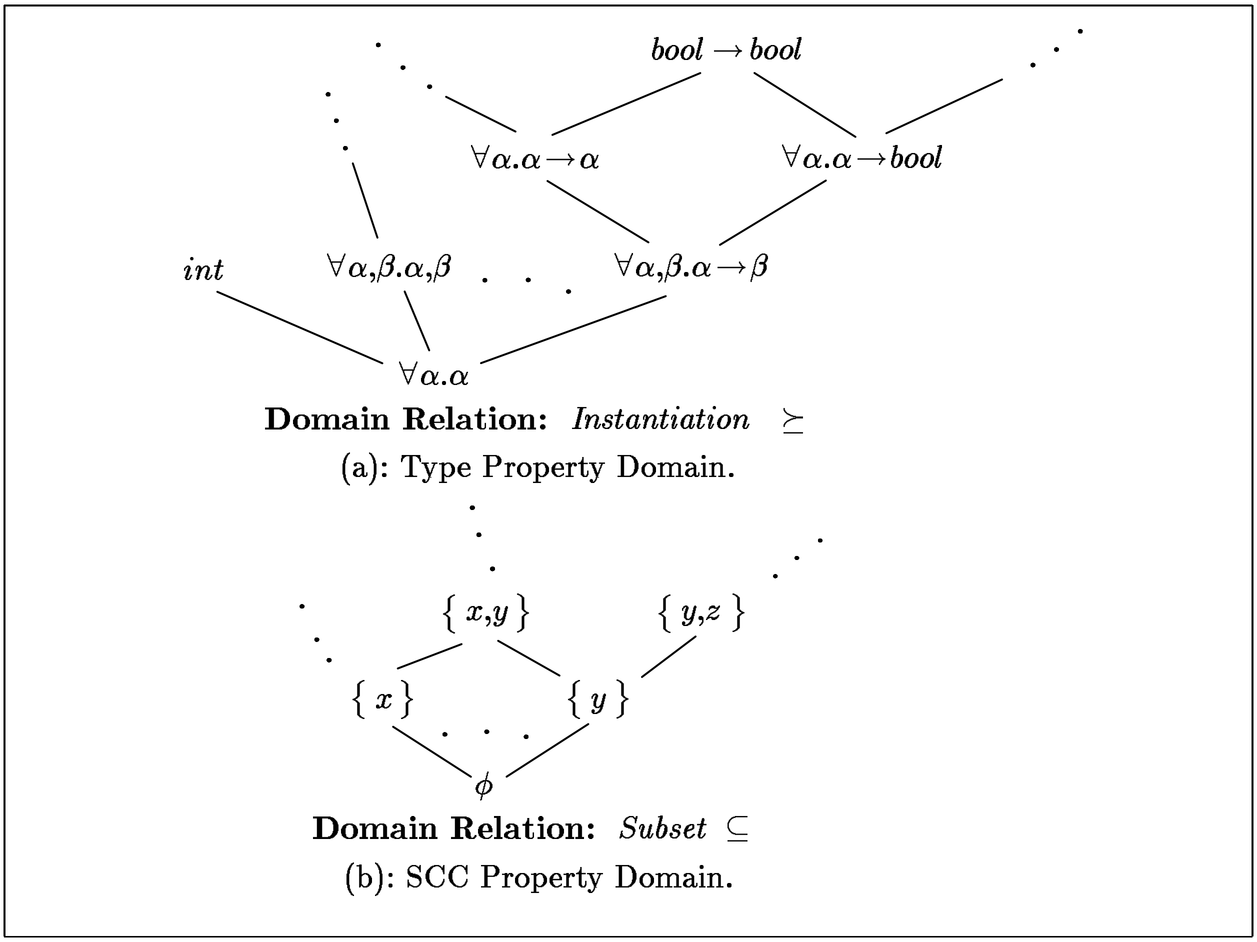


Figure 2: Compilation Properties used in the Incremental System.

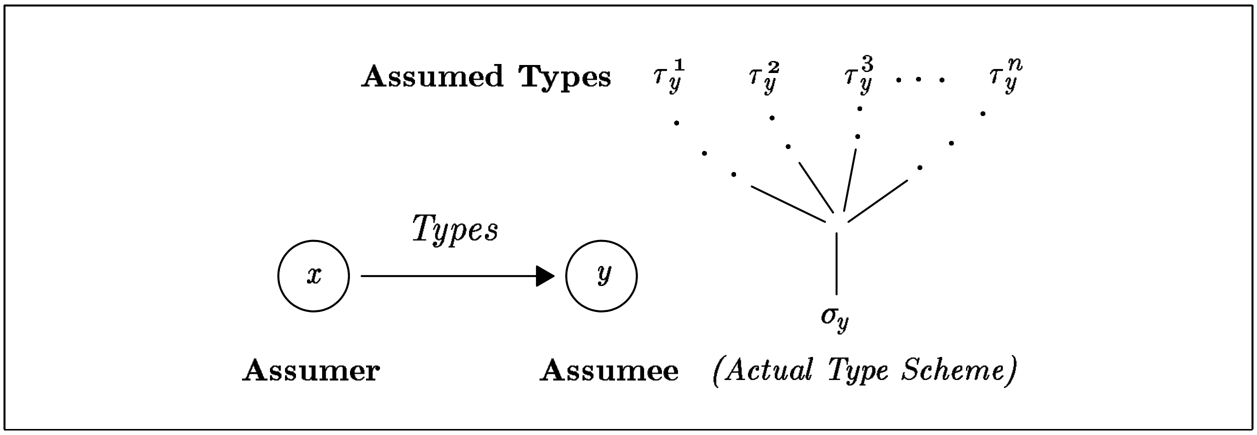


Figure 3: The Upper Bound Type Assumptions used in the Incremental System.

As a start, we consider only out-of-order compilations, *i.e.*, we will assume that editing of previously defined bindings is not permitted, though, new bindings can still be added incrementally. This is done to ensure monotonicity of the compilation environment and will be relaxed in a later section. We still allow the compiler to issue recompilations of previously encountered bindings if necessary, but the user is not allowed to edit them.

We maintain the following compilation properties and assumptions in this system.

Property – *Type* = (Type-Schemes, $\succeq$) (see Figure 2 (a)). This property records the type of all the identifiers. The "bottom" element of the domain is the most general type-scheme $\forall\alpha.\alpha$. As stated earlier, we denote the mapping of identifiers to their types in a Type-Environment (TE).

Property – *Strongly-Connected-Components* = (*powerset*(Identifiers), $\subseteq$) (see Figure 2 (b)). The *strongly connected component* (SCC) of an identifier is the set of all identifiers in the static call graph that are mutually recursive with the given identifier. We denote the mapping of the identifiers to their recursive sets in a SCC-Environment (SE).

Assumption – We define an assumption domain $\mathcal{M}$ called the *Upper-Bound-Type-Assumptions* that is used to record the actual type instance of each use of a free generic identifier in a top-level definition (see Figure 3).

$$\begin{array}{l}\text{COLLECT: } M_x = \{(y \mapsto \tau_y)\}, \quad \forall \text{ occurrences of } y \in (\mathit{free}(e_x) - SCC_x) \\ \text{CHECK: } \forall (y \mapsto \tau_y) \in M_x,\ TE(y) \succeq \tau_y\end{array} \tag{10}$$

These assumptions are called the upper bound type assumptions because in the partially ordered domain of type instances of the assumee's type-scheme, the recorded instances appear as an upper bound of useful instances used by the assumer. We collect a set of such assumptions for each assumer identifier defined on the left hand side of a top-level binding. The associated assumption check tests if these assumed types are valid instances of the latest type-scheme assigned to the assumee identifiers in some later type environment TE.

The upper bound type assumption checks ensure that the final type-scheme of an assumee identifier in the environment TE is at least as polymorphic as it was assumed when its assumer definition was compiled. If any of these checks fail, we may conclude that the assumee's type-scheme has changed significantly since the compilation of its assumer, and therefore, the assumer definition should be recompiled. For example, after the second redefinition of function `g` in section 4.2, its latest type `(int -> int)` can no longer instantiate the assumed type `(int -> (list int))` recorded in the upper bound type assumptions of the function `f`, which will then be flagged for recompilation by the system.

5.4 Incremental Algorithm

Our identifier namespace consists of the set $X = \{x_1, x_2, \ldots, x_n\} \cup \{\texttt{it}\}$ of identifiers bound in the LHS of the top-level bindings in a complete program as defined by equation 5. The initial property environments TE_0 and SE_0, assign the bottom elements of their respective domains to all the identifiers in the namespace.

$$TE_0 = TE_{lib} \cup \{x_k \mapsto \forall\alpha.\alpha\} \qquad \forall x_k \in X. \tag{11}$$

$$SE_0 = \{x_k \mapsto \phi\} \qquad \forall x_k \in X. \tag{12}$$

TE_0 also includes a standard library type environment TE_{lib} that maps all the standard library function identifiers, predefined operators etc. to their appropriate type-schemes. There are no free type-variables in this environment by assumption.

Our incremental type inference algorithm[11] W_1 appears in Figure 4. This algorithm corresponds to the *process-binding* function of Figure 1. With each invocation of W_1, we compute the compilation properties and the assumptions of a single top-level binding and accumulate them in the compilation environment.

[11] The notation $A\backslash_B$ used in the algorithm represents the mapping A with its domain restricted to elements other than those in set B.

```
Def W₁(TE_{i-1},SE_{i-1},B_i) =
    Let B_i be the binding 'x = e_x'.
    Construct the current call graph G_i using the free identifiers of B_i.
    Let SCC_x^i = {z_1, ..., x, ..., z_n} be the current SCC of x in G_i.

    SCC Environment:
        SE_i = SE_{i-1} \_{SCC_x^i} +{z_k ↦ SCC_x^i}          ∀z_k ∈ SCC_x^i.

    for each top-level identifier z_k ∈ SCC_x^i do
        Let e_{z_k} be the RHS expressions associated with z_k.
        Let β_{k1}, ..., β_{kn} be new type-variables.
        New-TE_k^i = TE_{i-1} \_{SCC_x^i} +{z_1 ↦ β_{k1}, ..., z_n ↦ β_{kn}}.
        (S_k^i, τ_{z_k}^i, A_k^i) = W'(New-TE_k^i, e_{z_k}).
    endfor.
```

$$U_i = \textit{Unify-Columns}\begin{pmatrix} S_1^i(\beta_{11}) & \cdots & S_1^i(\beta_{1k}) & \cdots & S_1^i(\beta_{1n}) \\ \vdots & & \vdots & & \vdots \\ S_n^i(\beta_{n1}) & \cdots & S_n^i(\beta_{nk}) & \cdots & S_n^i(\beta_{nn}) \\ \tau_{z_1}^i & \cdots & \tau_{z_k}^i & \cdots & \tau_{z_n}^i \end{pmatrix}$$

Type Environment:

$$TE_i = TE_{i-1} \setminus_{SCC_x^i} + \{z_k \mapsto close(U_i \tau_{z_k}^i)\} \qquad \forall z_k \in SCC_x^i.$$

Upper-bd Type Assumptions:

$$M_{z_k} = U_i A_k^i \setminus_{SCC_x^i} \qquad \forall z_k \in SCC_x^i.$$

Return (TE_i, SE_i).

Figure 4: The Incremental Algorithm W_1.

The first step in algorithm W_1 is to compute the static call graph[12] of the current binding $x = e_x$ (also denoted by B_i, corresponding to the i-th invocation of W_1), using the currently reachable nodes. Then, we partition this graph into its strongly connected components using standard graph theoretic techniques [1]. We record the strongly connected component SCC_x^i for the top-level identifier x within the SCC environment.

In our incremental system, we treat all the bindings contained within a SCC as a single *supernode* of the static call graph and type all of them together, each time any one of them changes. This is necessary in order to correctly identify and type the definitions that become mutually recursive incrementally. For instance, the examples of section 4.3 will all be typed correctly when the last definition in their strongly connected component is encountered. This may require keeping track of the actual code of each definition as it is compiled, which is not too difficult to maintain in the integrated editor-compiler environment of Id.

The set of bindings $z_k = e_{z_k}$ for each $z_k \in SCC_x^i$ are type-checked in two phases. First, we type-check each of the RHS expressions e_{z_k} independent of the types of other top-level identifiers of SCC_x^i. We also collect upper bound type assumptions for each z_k from its RHS. Then in the second phase, we unify the defined type $\tau_{z_k}^i$ obtained from typing each RHS, with the type of z_k as used by other bindings of the same SCC. We do this operation as one giant unification of all the terms corresponding to the same top-level identifier. This operation effectively simulates the task of the fixpoint constructor necessary to model recursive definitions in the batch system.

[12] A **Static Call graph** is a directed graph where each top-level binding is a node, and there is an edge from binding $x = e_x$ to binding $y = e_y$ in the graph whenever there is an occurrence of y inside e_x, or in other words, whenever x *uses* y.

```
Def W'(TE,e) =
  case e of
    x          ⇒ let
                    ∀α1 ··· αn. τ = TE(x).
                    β1, ..., βn be new type-variables.
                    τ' = {αi ↦ βi}τ.
                 in
                    (ID,τ',{(x ↦ τ')}).
    λx. e1     ⇒ let
                    β be a new type-variable.
                    TE' = TE + {x ↦ β}.
                    (S1, τ1, A1) = W'(TE', e1).
                 in
                    (S1, S1β → τ1, A1\{x}).
    e1 e2      ⇒ let
                    (S1, τ1, A1) = W'(TE, e1).
                    (S2, τ2, A2) = W'(S1(TE), e2).
                    β be a new type-variable.
                    S3 = U(S2τ1, τ2 → β).        (may fail)
                 in
                    (S3S2S1, S3β, S3(S2A1 ∪ A2)).
    let x = e1 in e2 ⇒
                 let
                    (S1, τ1, A1) = W'(TE, e1).
                    TE' = S1(TE) + {x ↦ close(S1(TE), τ1)}.
                    (S2, τ2, A2) = W'(TE', e2).
                 in
                    (S2S1, τ2, S2A1 ∪ A2\{x}).
  endcase.
```

Figure 5: Pseudo-code for the Inference Algorithm W'.

The algorithm W' that actually computes the type of each top-level RHS expression and records its type assumptions, is shown in Figure 5. It recursively collects all type instances generated for each use of a freely occuring identifier into a set of type assumptions for the current assumer. We will show some assertions about these collected assumptions in the next section. Apart from this extra book-keeping, the algorithm is exactly the same as the standard type inference algorithm W of [14]. Consequently, the theorems of soundness and completeness of the type computed by W with respect to the standard Hindley/Milner inference rules (see [4, 14]) are also valid for W'.

6 Correctness

In this section we outline the proof of correctness for our incremental type inferencing system. The detailed proofs of the lemmas and theorems appearing below are given in [5].

We need to show a correspondence between the incremental type inference strategy outlined in the previous section and the batch-mode Hindley/Milner type system that infers types for complete expressions. As given in equation 5, a program in the incremental system is an unordered sequence of bindings with the program query at the end. In the batch system, one would have to appropriately group mutually recursive bindings, reorder these groups bottom up according to their calling order, and process all of them together as a giant nested expression, with the program query occurring innermost. These two situations are contrasted

Incremental System	Batch System

$TE_0 = TE_{lib} + \{x_i \mapsto \forall\alpha.\alpha\}$:
$x_1 \; = \; e_1$
TE_1 :
$x_2 \; = \; e_2$
$\vdots$
TE_{i-1} :
$x_i \; = \; e_i$
TE_i :
$\vdots$
$\texttt{it} = e_{main}$
TE_f.
	$TE^t_{x_1} = TE_{lib}$:
	`let` $x_1 \; = \; e_1$ `in`
	$TE^t_{x_2} = TE^t_{x_3}$:
	`let` $x_2, x_3 \; = \; e_{23}$ `in`
	$\vdots$
	$TE^t_{x_i}$:
	`let` $x_i \; = \; e_i$ `in`
	$\vdots$
	$TE^t_{e_{\mathrm{main}}}$:
	e_{main}.
	$TE_t = TE^t_{e_{\mathrm{main}}} + \{\texttt{it} \mapsto \sigma_{e_{\mathrm{main}}}\}$.

Figure 6: A user session in the Incremental System and the Batch System.

in Figure 6. The incremental system corresponds to an interactive user session, while the batch system processes entire programs.

We intend to show that the type environment created in the batch system after typing the innermost program query "e_{main}" is exactly the same as the final environment reached in the incremental system after all the bindings have been processed and all their assumption checks have been passed in that environment, *i.e.*, $TE_f = TE_t$.

6.1 Some Results on Type Assumptions

Before we show the correspondence between the incremental system and the batch system, it is instructive to identify the importance of assumptions and assumption checks. The upper bound type assumptions collected *via* the algorithm W' serve as an exact type specification of the interface between the current top-level definition and other definitions. Thus, by recording them, we capture all the external type requirements of the given definition in the form of a type signature, which can be used later to reconstruct its type derivation in a different environment, simply by checking if all these assumptions were satisfied in that environment. This notion of assumption based type derivation tree is defined as follows.

Definition 3 *Given a typing $TE \vdash e : \tau$ and its associated derivation tree, we write $A^{TE} \vdash e : \tau$, or simply $A \vdash e : \tau$ (when TE is clear by context), as another typing representing the same derivation tree that explicitly records all type instances of its free (external) identifiers. $A^{TE} = assumptions(e)$ is the set of assumptions constructed inductively as follows:*

`Def` *assumptions(e)* =
  `case` e `of`

$$
\begin{array}{lll}
x & \Longrightarrow & \{(x \mapsto \tau)\} \\[1ex]
\lambda x.\, e_1 & \Longrightarrow & \texttt{let } TE' = TE + \{x \mapsto \tau_2\}\ ; \\
 & & \quad A^{TE'} \; = \; assumptions(e_1) \\
 & & \texttt{in} \\
 & & \quad A^{TE'} \backslash_{\{x\}} \\[1ex]
e_1\ e_2 & \Longrightarrow & \texttt{let } A_1^{TE} \; = \; assumptions(e_1)\ ; \\
 & & \quad A_2^{TE} \; = \; assumptions(e_2) \\
 & & \texttt{in}
\end{array}
$$

$$A_1^{TE} \cup A_2^{TE}$$

$$\begin{array}{lll} \texttt{let } x & = & e_1 \texttt{ in } e_2 \Longrightarrow \\ & & \texttt{let } TE' = TE + \{x \mapsto close(TE, \tau_1)\} \ ; \\ & & \quad A_1^{TE} \ = \ assumptions(e_1) \ ; \\ & & \quad A_2^{TE'} \ = \ assumptions(e_2) \\ & & \texttt{in} \\ & & \quad A_1^{TE} \cup A_2^{TE'} \setminus_{\{x\}} \end{array}$$

`endcase`.

Note that there may be several type instances recorded in A^{TE} for the same identifier x corresponding to its various occurrences within e. Also note that A^{TE} is not a type environment in itself, but it serves as a cumulative record of the use of the free identifiers of the expression e during the derivation of the typing $TE \vdash e : \tau$.

The range of an assumption set A^{TE} is a set of types, so we can apply substitutions to assumption sets just like we do to type environments. The only restriction is that substitutions do not apply to those type-variables in the range that were "closed" *via* the generalization operation during the typing of expression e. We call such type-variables $genvars(A^{TE})$. This restriction has the same effect as in the case of type environments, where by definition, substitutions do not apply to the bound type-variables in their range. The algorithm W' achieves this automatically by always using fresh type-variables to instantiate a previously closed type-variable. Thus, at each stage of the typing derivation, the substitutions used at that point never interfere with the previously closed type-variables.

The following lemma forms the backbone of the assumption based reasoning in the subsequent proofs for correctness of our incremental system.

Lemma 4 *Let $A^{TE} \vdash e : \tau$ be a typing corresponding to $TE \vdash e : \tau$ defined via definition 3. Then for any substitution S that does not involve $genvars(A)$ in its domain or range, $S(A^{TE}) \vdash e : S\tau$ is also a valid typing corresponding to $S(TE) \vdash e : S\tau$.*

The above lemma provides a way to use the assumption sets to obtain new correct typings from old typings for the same expression.

Now we come back to the algorithm W' (refer Figure 5) and show that the assumptions collected there actually correspond to the typing generated by the algorithm. The proof of this lemma is by structural induction on e and uses lemma 4.

Lemma 5 *If assumption set A is collected in the invocation $W'(TE, e) = (S, \tau, A)$ then $A \vdash e : \tau$ is a valid typing that corresponds to the typing $S(TE) \vdash e : \tau$ generated using structural induction on e under the type environment TE.*

The above lemma shows that the assumptions collected by algorithm W' are sound in the sense that they denote a valid typing.

6.2 Correctness of Algorithm W_1

The correctness of algorithm W_1 is established in two phases. First, we show that it is complete, *i.e.*, at all times the type environment resulting from type-checking a binding is a generalization of the final type environment TE_t obtained in the batch system.

Lemma 6 *If TE_t is the final type environment in the batch system with the type-scheme obtained for the final program query being assigned to the special variable "`it`", then for every $i \geq 0$, $TE_i \succeq TE_t$.*

The above lemma implies that starting with the most general type-scheme $\forall\alpha.\alpha$, the type-schemes of the top-level identifiers get constantly refined as their definitions are type-checked, but at each stage they remain complete with respect to their actual type-schemes. The proof of this lemma requires a similar statement regarding the SCC of top-level identifiers, that they grow monotonically from the empty set ϕ stablizing at their actual SCC.

The second step in proving the correctness of algorithm W_1 is to show that the process of incrementally refining the compilation environment eventually terminates, and when it does, it exactly corresponds to the compilation environment obtained in the batch system. As noted in section 5.2, we stop further compilation only when all the assumption checks collected during the various invocations to W_1 are passed successfully by the existing property-values in the current compilation environment. The assumption checks play an important role here in guaranteeing that only a sound compilation environment is acceptable, otherwise the process of incremental refinement continues. Termination is shown by proving that the assumptions checks are eventually passed by some environment.

Lemma 7 *In the sequence of type environments $TE_0, \ldots, TE_i, \ldots$ generated from the various invocations to W_1, there exists an environment TE_f (and every environment after that) that passes all the upper bound type assumptions (M) for all the top-level identifiers, i.e.,*

$$\forall x \in X,\ \forall(y \mapsto \tau_y^i) \in M_x,\quad TE_f(y) \succeq \tau_y^i$$

and moreover, $TE_t \succeq TE_f$.

The proof of this lemma uses the earlier lemmas 4 and 5 about the properties of assumptions collected. This also requires a similar lemma for the soundness of SCC environment.

The soundness of the types obtained in our incremental system depends on whether all the upper bound type assumptions checks have been passed or not. We state this in the form of a termination strategy.

Termination Strategy 1 *Assuming that all the top-level definitions in the given program are compiled at least once, the incremental compilation system terminates when all the upper bound type assumptions (M) for each of the top-level identifiers pass their check in the latest compilation environment.*

Finally, we state the correctness theorem.

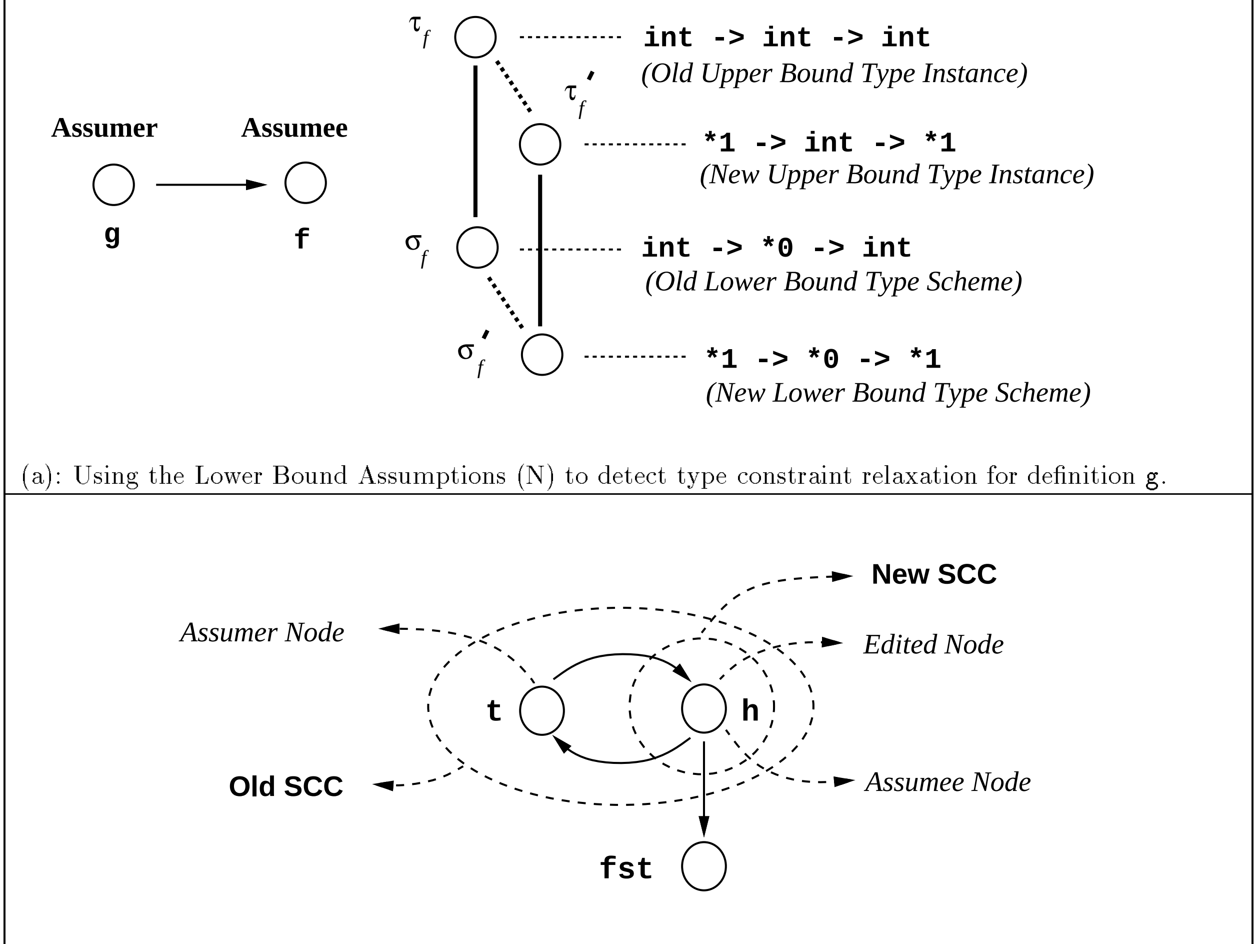
Theorem 8 *Given a complete and type-correct program, when the incremental system terminates, it has computed exactly the same type and SCC environments as computed in a batch system.*

Proof: The incremental system starts in an empty compilation environment (TE_0 and SE_0) as given by equations 11 and 12. So the completeness lemma 6 is applicable. The termination strategy ensures that the system terminates in an environment (TE_f and SE_f) that satisfies the soundness lemma 7. Combining the result of these lemmas we straightaway obtain $SE_f = SE_t$ and $TE_f = TE_t$. □

7 Extensions to Algorithm W_1

7.1 Complete Programs

Uptil now we dealt with a given set of top-level definitions supplied incrementally that were assumed to form a complete program. Actually, the notion of a complete program depends on the program query. Only those definitions that are called by the program query, directly or indirectly, need to be supplied and checked for consistency. The status of other definitions that are unrelated to the given query, is unimportant. Therefore, in the incremental system, we perform the consistency checks only for the minimal set of definitions that are self contained with respect to the given program query. This set can be computed by looking at the minimal call graph rooted at the given query, that does not have any dependency edges going out of it. This can enormously reduce the number of definitions that need to be checked.



(a): Using the Lower Bound Assumptions (N) to detect type constraint relaxation for definition `g`.

(b): Using the SCC Assumptions (Q) to detect reduced SCCs for definition `t`.

Figure 7: Examples of additional assumptions being used to detect Constraint Relaxation during editing.

7.2 Constraint Relaxation

The other problem that we have not yet addressed is editing that leads to constraint relaxation. The completeness lemma 6 implies that the compilation environment of the incremental system is never allowed to be more constrained than the final environment of the batch system. This may not always be the case due to editing when some constraints on the bindings may get relaxed as shown in section 4.4. It is the user's responsibility to recompile a definition after editing it, but it is the compiler's responsibility to propagate those changes throughout the rest of the program and warn the user accordingly while maintaining soundness and completeness of the derived types.

We can detect these constraint relaxations by using some additional property assumptions and checks as defined below. We can collect these extra assumptions along with the upper bound type assumptions already collected in algorithm W_1.

Assumption – We define an assumption domain $\mathcal{N}$ called the *Lower-Bound-Type-Assumptions* that is used to record the actual type-schemes of free generic identifiers used within a top-level definition.

$$\begin{array}{l} \text{COLLECT: } N_x = \{y \mapsto TE(y)\}, \qquad \forall y \in (free(e_x) - SCC_x) \\ \text{CHECK: } \forall (y \mapsto \sigma_y) \in N_x,\ \sigma_y \succeq TE(y) \end{array} \tag{13}$$

These assumptions are called the lower bound type assumptions because the recorded assumee type-scheme is a lower bound for the assumee type instances used by the assumer according to the partial order of the type domain. Note that the assumption check uses the same subsumption test as for the upper bound type assumptions, but in the reverse direction. The check fails if the assumee type-scheme becomes more polymorphic than it was assumed to be when the assumer was compiled. This enables the assumer to be flagged for recompilation in order to adjust its type according to the latest assumee type-scheme.

Assumption – We also define an assumption domain $\mathcal{Q}$ called *SCC-Assumptions* that is used to record the SCC of each identifier used directly by another identifier from within its own SCC.

$$\begin{array}{ll} \text{COLLECT: } Q_x = \{y \mapsto SCC_y\}, & \forall y \in (free(e_x) \cap SCC_x) \\ \text{CHECK: } \forall (y \mapsto SCC_y) \in Q,\ SCC_y = SE(y) & \end{array} \tag{14}$$

The consistency check in this case is a test of (set) equality. This ensures that if the SCC of one of the identifiers changes, then all identifiers that were excluded from its old SCC during this change will fail this check and will be flagged for recompilation.

To see how these additional assumptions help in detecting constraint relaxation, we apply them to the examples given earlier in section 4.4.

The situation for the first example of section 4.4 is pictorially depicted in Figure 7 (a). The function `g` uses function `f` with type instance `(int -> int -> int)` derived from its initial type-scheme `(int -> *0 -> int)` which is recorded in the lower bound assumption set of `g`. After function `f` is edited, its new type-scheme becomes `(*1 -> *0 -> *1)` which fails the lower bound assumption check against the type-scheme recorded in `g`'s lower bound assumption set. Therefore, the system will be able to flag the recompilation of function `g`, so that it may benefit from the relaxed type of function `f`. Note that this type relaxation of `f` can not be detected *via* the upper bound type assumptions of `g` because `f`'s type instance recorded there remains an instance of the relaxed type-scheme of `f`.

Looking at the second example given in section 4.4 (see Figure 7 (b)), the function `t` records the SCC of the function `h`, {`h,t`}, in its SCC assumption set. After `h` is edited, its new SCC, {`h`}, fails the SCC assumption check against the earlier value recorded in the SCC assumption set of `t`. Thus, the system will be able to flag the recompilation of function `t` which will relax its type and SCC properties to their correct values.

Our new termination strategy is now only slightly more complex.

Termination Strategy 2 *To test for termination of the extended incremental system, we execute the following steps.*

1. *Build a static call graph with the query binding "*`it` $= e_{main}$*" as the root.*
2. *Check if any leaf identifier from the above call graph has not yet appeared for compilation.*
3. *Check if the upper bound type assumptions (M), the lower bound type assumptions (N), and the SCC assumptions (Q) for each identifiers in the call graph are all satisfied in the latest compilation environment.*
4. *If all the above tests are passed, then the incremental system terminates.*

Our incremental type inference system is now complete. We use all the assumption domains $\mathcal{M}$, $\mathcal{N}$, and $\mathcal{Q}$, and the termination strategy 2 in the overall plan of Figure 1, while using the algorithm W_1 to process each top-level binding. We can type-check a set of definitions incrementally, allowing editing of earlier definitions. The additional assumption domains $\mathcal{N}$ and $\mathcal{Q}$ make sure that the types inferred are both sound and complete even with arbitrary editing.

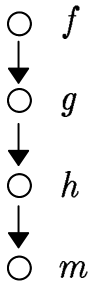
```
def f x = g x;

def g x = h x;

def h x = m x;

def m x = x:nil;
```

(a)



(b): Static Call Graph.

```
def f x = g x;

def g x = (h x):nil;

def h x = (m x):nil;

def m x = x:nil;
```

(c)

Figure 8: Example program to illustrate the high number of Recompilations.

8 Complexity and Optimizations

8.1 Complexity of the Incremental System

Apart from the inherent complexity of the Hindley/Milner type system, which is known to be exponential [6], our incremental system incurs book-keeping and recompilation overheads. The book-keeping cost can be included within the regular cost of compiling a program. Therefore, we will only focus on the number of recompilations necessary in our incremental system to arrive at the correct typings.

Incremental changes in the program potentially affect all the topologically preceding definitions starting from the point of change in the call graph. Our mechanism of maintaining upper and lower bound assumptions attempts to reduce this set of affected definitions. When the definition of an identifier appearing later in the incremental sequence of compilations falls within the assumed bounds, no recompilation is flagged. Even when these assumptions fail, we can localize the retyping to just the assumer identifier and its SCC instead of the whole program. But, in the worst case, each such recompilation can give rise to more failures in the topologically preceding SCCs and we may end up recompiling the whole program.

As an example, consider the set of definitions given in Figure 8 (a). The call graph in this case is a simple chain as shown in (b). The upper bound type assumptions of the identifier `h` fail the assumption checks as soon as `m` is defined, and `h` has to be recompiled. The recompilation of `h` propagates the type information of `m` onto `h` and causes the failure of assumption checks of `g` and so on. This example shows that the cost of an incremental change to the program could be linear in its size.

Recompiling a definition as soon as some of its assumptions fail, may not be the best strategy. The definitions given in Figure 8 (c) will incur quadratic number of recompilations following this strategy, because every time a new definition is encountered, the upper bound type assumptions of all the preceding definitions fail. It is possible to amortize the cost of recompilation over several assumption failures by adopting a lazy recompilation strategy. If we postpone all recompilations until all the definitions have been encountered, a single sweep of recompilations in the reverse order will settle all the types to their correct values. We have adopted this lazy strategy in Id by postponing all recompilations until we are ready to evaluate a program query (see Figure 1).

8.2 Optimizations

It is possible to further reduce the cost of an incremental change to the program, if it does not affect the properties of other nodes. In our current scheme, whenever a node is flagged for recompilation, we recompile all the definitions in its latest SCC. This is necessary when the new SCC of the node is different from its old one. But this may be wasteful if the editing changes do not affect the SCC or the type properties of the node. Indeed, the only computation really necessary in such cases is the compilation of the edited node itself.

Even when the type of the edited node is different but its SCC is the same, it may not be necessary to recompile all the other definitions in its SCC. This is because type-checking a given definition does not use any information about other definitions in its SCC (apart from the fact that they are all in the same SCC) until the latter part of algorithm W_1 when we unify all the assumed types of the definitions with their computed types in a unification matrix (see Figure 4). Therefore, if we save the assumed types of SCC identifiers while typing a given definition, we can use them directly in the latter part of the W_1 algorithm when some other identifier from the same SCC is being recompiled. The details of these optimizations and a modified inference algorithm W_2 appear in appendix A.

9 Conclusions

9.1 Summary

We have shown that it is possible to obtain correct Hindley/Milner typings for programs of a ML-like language when placed in an incrementally compiled environment geared towards easy editing, debugging and testing. Our approach is guided by Lisp-like environments that offer definition level incrementality and dynamic linking. A single toplevel definition forms a convenient unit of compilation and inter-unit consistency maintenance both in terms of maintaining modularity and minimizing book-keeping overhead.

We have achieved this incremental flexibility by maintaining the interface of a top-level definition with respect to the other definitions in a consistent fashion. We described a general mechanism of incremental property collection and consistency maintenance, which was applied in our case to maintaining the types and the strongly connected components of top-level definitions of a program. We took advantage of the inherent polymorphism of the type system by keeping only the upper and lower bounds of type usage and maintaining the definitions within those bounds.

We should mention that the automatic, incremental book-keeping mechanism we described in the preceding pages is independent of the program granularity it is applied to. It could very well be applied to program modules consisting of groups of definitions, or to complete files, or at a finer grain, to individual subexpressions inside a top-level definition.

9.2 Comparison with Related Work

Some other systems [7, 13, 15] also provide incremental compilation. The GLIDE system [15] comes closest to our philosophy of incremental type inference because it is also based on the ideas presented by Nikhil in [10], but it differs significantly in approach.

The GLIDE system employs an automatic, lazy loading strategy at run-time whereby definitions are automatically compiled and loaded into the run-time environment only as needed by the currently executing expression. It also does retyping of old definitions on the fly, if necessary. In sharp contrast, our system has clearly distinguishable phases of compilation, loading, consistency checking, and execution. Each phase is explicitly initiated by the user which gives him/her complete control over scheduling of various definitions through the individual phases.

The lazy loading mechanism of GLIDE has the advantage of touching only those definitions that are actually needed at run-time, but it forces the system to perform all the necessary consistency checks and resulting retypings immediately as each new definition is loaded. This must be done in order to detect any fresh type errors caused by the changed or new definition before the execution can resume. As pointed out in section 8.1, such a strategy may end up in quadratic number of retypings for programs resembling the example in Figure 8 (c). The GLIDE system attempts to reduce the work involved in such retypings by sharing type information between multiple retypings of the same definition.

In contrast, our system permits early loading of definitions and puts all retyping under explicit user control. The user is guided by the errors reported during consistency checking in appropriately scheduling the retypings so that no definition needs to be retyped more than once. Note that the consistency checks now have to be performed for each definition in the static call graph of the query expression rather than the definitions actually used in its execution.

Probably the most important difference between our system and the GLIDE system is in the actual framework for consistency checking and maintenance. We have been able to formalize an independent system of incremental property maintenance, and have used it successfully to maintain type and SCC properties of top-level definitions. The GLIDE system, on the other hand, has a specific incremental type inference system built into it. Our type consistency checks use the inherent polymorphism of the language to advantage and avoid retyping where simple syntactic interference checks as used in GLIDE may not be able to do so. And finally, we collect and maintain incremental type information only at the granularity of top-level definitions and no record is kept for internal subexpressions. The GLIDE system, on the other hand, maintains type information at all subexpressions so that it may be reused during retyping.

9.3 Status

Finally, we should point out that the incremental type inference system for Id described in this paper has been implemented in the Id compiler and has been in use for the last two years.

10 Acknowledgements

We would like to express our thanks to Zena Ariola, Boon Ang, Alejandro Caro, and Steve Glim, from the Computation Structures Group at Laboratory for Computer Science, MIT, for their insightful comments and help in proof-reading this paper.

A Optimizations for Algorithm W_1

As mentioned in section 8.2, we can optimize the algorithm W_1 by saving the type instances used by each top-level identifier and using them later while typing other top-level identifiers from the same SCC. Essentially, we need to save all the type information necessary to reproduce the unification matrix in the latter half of algorithm W_1 (see Figure 4). We will use our incremental property maintenence system to record this information as compilation properties of the corresponding top-level identifiers. Note that these properties are "local", *i.e.*, they are independently computed for each identifier in a SCC and do not use any information about other identifiers from the same SCC. Therefore, we do not need to maintain any assumption checks for them. We maintain the following two local properties.

Local property – *Local Type* = (Types, =). For each identifier x, we record its local type τ_x^i computed in the invocation of W'. We save this property in a map from identifiers to types called Local-Type-Environment WE.

Local property – *Local Assumptions* = (Local-Type-Environments, =). For each identifier x, its local assumption set A_x^{local} is a map from the identifiers in its SCC to their types as inferred from the invocation of W'. We collect this property in Local-Assumption-Environment AE.

The updated algorithm W_2 appears in Figures 9 and 10, where we have also incorporated the collection of the SCC and the lower bound type assumptions introduced in section 7. The algorithm now operates in two phases.

The first phase computes the SCC of the given definition and compares it with its earlier value. If the SCC has changed then we proceed as before, compiling each of the definitions belonging to the new SCC afresh

Def $W_2(TE_{i-1}, SE_{i-1}, B_x) =$

PHASE I:

Let B_x *be the binding* '$x \ = \ e_x$'.

Construct the current call graph G_i *using the free identifiers (FId) property.*

$SCC^i_x = \{z_1, \ldots, x, \ldots, z_n\}$ *be the current SCC of* x *in* G_i.

if $SCC^i_x \neq SE_{i-1}(x)$ **then**

SCC Environment: $SE_i = SE_{i-1} \setminus_{SCC^i_x} + \{z_k \mapsto SCC^i_x\} \qquad \forall z_k \in SCC^i_x$.

for each *top-level identifier* $z_k \in SCC^i_x$ **do**

Let e_k *be the RHS expressions associated with* z_k.

SCC Assumptions: $Q_{z_k} = \{y \mapsto SCC^i_x\} \qquad \forall y \in (free(e_{z_k}) \cap SCC^i_x)$.

Let $\beta_{k1}, \ldots, \beta_{kn}$ *be new type-variables.*

$New\text{-}TE^i_k = TE_{i-1} \setminus_{SCC^i_x} + \{z_1 \mapsto \beta_{k1}, \ldots, z_n \mapsto \beta_{kn}\}$.

$W'(New\text{-}TE^i_k, e_k) = (S^i_k, \tau^i_{z_k}, A^i_k)$.

Local Assumptions: $A^{local}_k = \{z_1 \mapsto S^i_k\beta_{k1}, \ldots, z_n \mapsto S^i_k\beta_{kn}\}$.

endfor.

Local Assumption Environment:

$AE_i = AE_{i-1} \setminus_{SCC^i_x} + \{z_k \mapsto A^{local}_k\} \qquad \forall z_k \in SCC^i_x$.

Local Type Environment:

$WE_i = WE_{i-1} \setminus_{SCC^i_x} + \{z_k \mapsto \tau^i_{z_k}\} \qquad \forall z_k \in SCC^i_x$.

else

SCC Assumptions: $Q_x = \{y \mapsto SCC^i_x\} \qquad \forall y \in (free(e_x) \cap SCC^i_x)$.

Let $\beta_1, \ldots, \beta_n$ *be new type-variables.*

$New\text{-}TE^i_x = TE_{i-1} \setminus_{SCC^i_x} + \{z_1 \mapsto \beta_1, \ldots, z_n \mapsto \beta_n\}$.

$W'(New\text{-}TE^i_x, e_x) = (S^i_x, \tau^i_x, A^i_x)$.

Local Assumptions: $A^{local}_x = \{z_1 \mapsto S^i_x\beta_1, \ldots, z_n \mapsto S^i_x\beta_n\}$.

Local Assumption Environment: $AE_i = AE_{i-1} \setminus_{\{x\}} + \{x \mapsto A^{local}_x\}$.

Local Type Enironment: $WE_i = WE_{i-1} \setminus_{\{x\}} + \{x \mapsto \tau^i_x\}$.

endif.

Figure 9: PHASE I of the Updated Incremental Algorithm W_2.

and accumulating their properties. We also record the locally inferred type of each identifier and its local type assumptions about the other identifiers in its SCC for future use. If the SCC has not changed from its earlier value, then it implies that only the current binding needs to be compiled and we can use previously saved local properties of the other identifiers of that SCC in the second phase, instead of computing all of them afresh.

The second phase, as before, constructs a unification matrix using the freshly computed or previously saved local properties of all the identifiers of the SCC and updates their type properties. It also records the upper and the lower bound type assumptions for the identifiers compiled in the first phase.

References

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

[2] Andrew W. Appel and David B. MacQueen. *Standard ML Reference Manual*. Princeton University and AT&T Bell Laboratories, Preliminary edition, 1989. Distributed along with the Standard ML of New Jersey Compiler.

PHASE II:

$$U_i = \textit{Unify-Columns}\begin{pmatrix} AE_i(z_1)(z_1) & \cdots & AE_i(z_1)(z_k) & \cdots & AE_i(z_1)(z_n) \\ \vdots & & \vdots & & \vdots \\ AE_i(z_n)(z_1) & \cdots & AE_i(z_n)(z_k) & \cdots & AE_i(z_n)(z_n) \\ WE_i(z_1) & \cdots & WE_i(z_k) & \cdots & WE_i(z_n) \end{pmatrix}$$

Type Environment:

$TE_i = TE_{i-1} \setminus_{SCC_x^i} + \{z_k \mapsto close(U_i\, WE_i(z_k))\} \quad \forall z_k \in SCC_x^i.$

if *only x was type-checked in PHASE I* **then**

Upper Bound Type Assumptions: $M_x = U_i A_x^i \setminus_{SCC_x^i}$.

Lower Bound Type Assumptions:

$N_x = \{y \mapsto TE_{i-1}(y)\} \quad \forall y \in (free(e_x) - SCC_x^i).$

else

Upper Bound Type Assumptions: $M_{z_k} = U_i A_k^i \setminus_{SCC_x^i} \quad \forall z_k \in SCC_x^i.$

Lower Bound Type Assumptions:

$N_{z_k} = \{y \mapsto TE_{i-1}(y)\} \quad \forall y \in (free(e_k) - SCC_x^i),\ \forall z_k \in SCC_x^i.$

endif.

Return (TE_i, SE_i).

Figure 10: PHASE II of the Updated Incremental Algorithm W_2.

[3] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A Simple Applicative Language: Mini-ML. In *Proceedings of the ACM Symposium on LISP and Functional Programming*, pages 13–27, August 1986.

[4] L. Damas and R. Milner. Principle Type Schemes for Functional Programs. In *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[5] Shail Aditya Gupta. An Incremental Type Inference System for the Programming Language Id. Technical Report MIT/LCS/TR-488, Laboratory for Computer Science, 545 Technology Square, MIT, Cambridge, MA 02139, November 1990. First published as the author's Master's thesis.

[6] Harry G. Mairson. Deciding ML Typability is Complete for Deterministic Exponential Time. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages*, pages 382–401, January 1990.

[7] Lambert Meertens. Incremental Polymorphic Type Checking in B. In *Proceedings of the 10th ACM Symposium on Principles of Programming Languages*, January 1983.

[8] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[9] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Massachusetts, 1990.

[10] Rishiyur S. Nikhil. Practical Polymorphism. In *Proceedings of the Conference on Functional Programming Languages and Computer Architectures, Nancy, FRANCE*, volume 201 of *Lecture notes in Computer Science*. Springer-Verlag, September 1985.

[11] Rishiyur S. Nikhil. Id Version 90.0 Reference Manual. Technical Report CSG Memo 284-1, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, July 1990.

[12] Rishiyur S. Nikhil, P. R. Fenstermacher, J. E. Hicks, and R. P. Johnson. *Id World Reference Manual.* Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, revised edition, November 1989.

[13] Aaron Sloman and the Poplog Development Team. POPLOG V14 - A portable, multi-language, interactive software development environment with X11R4 interface. School of Cognitive and Computing Sciences, Sussex University, Brighton, BN1 9QH England, December 1990. (Overview description obtained *via* personal communication).

[14] Mads Tofte. *Operational Semantics and Polymorphic Type Inference*. PhD thesis, University of Edinburgh, Department of Computer Science, 1988. Also published as ECS-LFCS-88-54.

[15] Ian Toyn, Alan Dix, and Colin Runciman. Performance Polymorphism. In *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 325–346. Springer-Verlag, 1987. Proceedings of the FPCA Conference held in Portland, Oregon, 1987.

[16] David A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proceedings of the Conference on Functional Programming Languages and Computer Architectures, Nancy, FRANCE*, volume 201 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1985.

[17] David A. Turner. An Overview of Miranda. *SIGPLAN Notices*, 21(12):158–166, December 1986.