*J. Functional Programming* 6 (1) 111–141, January 1996 © 1996 Cambridge University Press 111

# *Weak polymorphism can be sound*

JOHN GREINER

*School of Computer Science, Carnegie Mellon University,*
*5000 Forbes Ave., Pittsburgh, PA 15213-3891, USA*

**Abstract**

The weak polymorphic type system of Standard ML of New Jersey (SML/NJ) (MacQueen, 1992) has only been presented as part of the implementation of the SML/NJ compiler, not as a formal type system. As a result, it is not well understood. And while numerous versions of the implementation have been shown unsound, the concept has not been proved sound or unsound. We present an explanation of weak polymorphism and show that a formalization of this is sound. We also relate this to the SML/NJ implementation of weak polymorphism through a series of type systems that incorporate elements of the SML/NJ type inference algorithm.

**Capsule Review**

The problem of safely incorporating assignment into a Hindley–Milner polymorphic type system has received considerable attention for more than a decade. Numerous solutions have been proposed and carefully studied. Standard ML of New Jersey, one of the most widely used implementations of Standard ML, employs a method known as *weak polymorphism.* But despite the inclusion of weak polymorphism in the New Jersey implementation for many years, weak polymorphism has received little formal study and is not generally understood by ML programmers.

This paper presents a long-overdue formal study of weak polymorphism. The paper exposes the concepts underlying weak polymorphism through a number of examples, codifies these concepts in a formal type system, and presents a proof that this type system is sound. Several refinements yield type systems that closely model the behaviour of the Standard ML of New Jersey implementation. This paper should be valuable both to type system designers and to programmers seeking to understand the intricacies of weak polymorphism.

## 1 Background

A reference cell is an assignable store location and is the primary imperative feature of Standard ML (SML) (Milner *et al.*, 1990). However, it is well known that its Hindley-Milner type system (Milner, 1978) is unsound if reference types are polymorphically generalized in the usual manner. For example, in the following expression, polymorphically generalizing the type of `ref nil` allows that reference cell to be used as both an `int list ref` and a `bool list ref`:

```
let val a = ref nil
in
    a := [1]; not (hd(!a))
end
```

This code stores a list of integers in the cell, but then reads a list of booleans from it. Such inconsistencies must not be allowed by the type system.

While it is unsound to polymorphically generalize reference types, it is not necessarily unsound to generalize function types involving reference types. A simple example is the following expression:

```
let val ref' = fn x => ref x
in
    ref' 1; ref' true
end
```

The function `ref'`, which creates reference cells, can safely be used at both the types `int -> int ref` and `bool -> bool ref`.

A number of type systems have been proposed to allow code such as the latter example while preserving soundness (Damas, 1985; Hoang *et al.*, 1993; Leroy and Weis, 1991; Talpin and Jouvelot, 1992a, 1992b; Wright, 1992). Of particular interest for this paper are those of Tofte (1988), which is used in the definition of SML, and MacQueen (1992), which is used in the New Jersey implementation of SML (SML/NJ).

In the standard Hindley–Milner type system, generalization is allowed on all free type variables not occurring in the *variable type assumption*, which is a mapping from variables to type schemes. As shown by Tofte, if references are added to the language, the type system must also have a *location type assumption* assigning a type to each location of the store. The unsoundness of the first example above is a result of generalizing a type variable occurring free in the location type assumption. Since neither the store nor location type assumption can be known statically, a safe approximation must be made to determine those type variables which may occur in the location type assumption and should not be generalized.

To provide such a safe approximation, Tofte introduces a distinction between two classes of type variables, called *applicative* and *imperative*. Applicative type variables are not used with reference types and can always be polymorphically generalized if they are not free in the variable type assumption. Imperative type variables are used to statically track values that may be placed in reference cells and can be generalized only if the evaluation of the *let*-bound expression does not lead to the creation of a reference cell. Determining when an expression leads to the creation of a cell is undecidable, so Tofte conservatively assumes that expressions that are not syntactic values may create cells. He calls these expressions *expansive*. Thus imperative type variables can only be polymorphically generalized if the expression is non-expansive.

The expansiveness criterion treats the above examples appropriately. The first is not typable since the expression `ref nil` is expansive, and the imperative type variable `'_a` in its type `'_a list ref` is not generalized. The second is typable

https://doi.org/10.1017/S0956796800001593 Published online by Cambridge University Press

since the definition of `ref'` is non-expansive, and its type is generalized and then instantiated to both `int -> int ref` and `bool -> bool ref`.

However, this static analysis is overly conservative. In the example

```
let val ref2 = fn x => fn y => ref x
in
    let val ref1 = ref2 nil
    in
        true :: !(ref1 ()); 1 :: !(ref1 ())
    end
end
```

the type of `ref1` is not generalized because `ref2 nil` is considered expansive. Since there is no generalization, `ref1` cannot have both the type `unit -> bool ref` and `unit -> int ref`, and the expression is not typable. However, allowing this example is sound, since a different reference cell is created for each application of `ref1`.

One method to improve upon Tofte's system is to track not only which values might be placed in reference cells, but *when* the cells are created. This additional information can then be used for a more accurate definition of expansiveness, which is the essence of MacQueen's *weak* polymorphic types.

## 2 Weak polymorphic types

Weak polymorphism expands on Tofte's distinction of type variables. To produce a better static analysis of what values may be in reference cells, each type variable has a *strength*, which is an integer or positive infinity. Type variables of infinite strength correspond to Tofte's applicative type variables, whereas those of finite strength correspond to imperative type variables. *Non-critical* type variables, those of positive strength, may be generalized, but *critical* variables may not. A more detailed comparison of the two systems is found in section 7.

The strength $s$ of a type variable $\alpha$ in the type of an expression $e$ indicates to how many arguments $e$ may be applied before a cell of a type involving $\alpha$ might be created. More precisely, if $0 \leq i < s$, and no expression in $e_1, \ldots, e_i$ uses reference cells, then the expression $e\ e_1 \ldots e_i$ will not create such a cell. In particular, if $s = 0$, then expression $e$ might create such a cell. Following SML/NJ, we assume that each instance of the same type variable in a type has the same strength.

Using these definitions, we now look at a series of examples which point out the key ideas of the type system formalized in section 3. The type of a reference cell should never be generalized, and thus must contain only critical type variables. So we have

```
ref nil : '0a list ref
```

where `0` is the strength of type variable `'a`. Purely functional terms have types of infinite strength, as in

```
fn x => x : 'a -> 'a
```

(The SML/NJ convention is that infinite strengths are not printed.) Placing an ex-

https://doi.org/10.1017/S0956796800001593 Published online by Cambridge University Press

pression inside the syntactic context of an abstraction increments the strengths, since they count the number of applications needed until a reference may be created:

```
fn x => ref nil : 'b -> '1a list ref
fn x => ref x : '1a -> '1a ref
```

Conversely, placing an expression in the function position of an application context decrements the strengths:

```
(fn x => ref x) nil : '0a list ref
```

An important idea to point out from these examples is that the strengths of type variables in an expression's type depend on the syntactic context of that expression.

The most complicated case is when an expression occurs as the argument of an application. The type and strengths of the function provide little information about how its argument is used in its body. As a result, if the argument's type involves weak type variables, a conservative approximation must be made. In general, the function may in turn apply its argument to multiple arguments, where each application corresponds to a decrement in strengths. Statically, the conservative assumption is made that enough applications are performed for a reference cell to be produced. For example, the strength of `'a` is 2 in

```
(fn x => fn y => ref x) : '2a -> 'b -> '2a ref
```

but `'a` must be critical when this expression is used as an argument in

```
(fn f => f nil nil) (fn x => fn y => ref x) : '0a list ref
```

While these examples do not use negative strengths, following these principles does lead to that possibility. Similar to the previous example, we have

```
(fn f => f nil) (fn x => fn y => ref x) : 'b -> '0a list ref
```

It would be sound to have the strength of `'a` be 1 here, but the analysis is overly conservative. Providing another argument decrements these strengths:

```
((fn f => f nil) (fn x => fn y => ref x)) nil : '~1a list ref
```

Usually, negative strengths occur only when type-checking subexpressions of the original expression, as in

```
ref (fn z => z) : ('0a -> '0a) ref
```

The strength of `'a` when type-checking `z` is $-1$, which is then incremented by the abstraction.

By typing applications less conservatively, SML/NJ avoids negative strengths at the top-level in most cases. But Version 0.66, which closely follows this motivation, assigns negative strengths in the following example:

```
(let val x = ref (fn z => z) in fn y => x end) ()
: ('~1a -> '~1a) ref
```

The `let` expression has type `unit -> ('0a -> '0a) ref` since a reference is created, and the application to `()` decrements the strengths one more.

Weak polymorphism has been developed by MacQueen within the type inference algorithm of SML/NJ. Only this algorithm has served as the definition of the type system, and several versions of the algorithm have been shown unsound. Each had problems which could be ascribed to implementation details, but the concept has not been proven sound or unsound. Furthermore, key ideas of the algorithm, such as application's conservative approximation, have not been widely known, so the system has been poorly understood even by skilled SML/NJ programmers.

This paper addresses these problems. In section 3, we incorporate the ideas outlined in this section into a formal type system, the soundness of which is outlined in section 4 and proved in full in the Appendix. In section 5, we show how this formalism relates to more algorithmic formalisms similar to that of SML/NJ. In sections 6 and 7, we compare the formalisms to SML/NJ, and to related work including Tofte's, respectively.

## 3 A declarative formalism – $\lambda\Sigma$

This section presents a formal definition of the static semantics of an SML-like language $\lambda\Sigma$ as outlined by section 2.

The expressions of this language are defined by the following grammar:

$$\begin{array}{rcl} x \in \mathit{variables} & = & \text{a countably infinite set} \\ l \in \mathit{locations} & = & \text{a countably infinite set} \\ e \in \mathit{expressions} & ::= & x \mid l \mid () \mid \mathit{ref}\ e \mid !e \mid e_1 := e_2 \mid \\ & & \mathit{fn}\ x \Rightarrow e \mid e_1\ e_2 \mid \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \end{array}$$

Locations are the formal equivalents of reference cells and are allowed as expressions to simplify the dynamic semantics and its correspondence to the static semantics. Expressions that are $\alpha$-equivalent are identified, and capture-avoiding expression substitution, $[e'/x]e$, is defined in the usual manner.

The static and dynamic semantics use several kinds of finite mappings. An empty mapping is denoted by $[\,]$. The extension of mapping $X$ to an additional domain element $d$ is denoted by $X[d \mapsto r]$, or $X[d : r]$ for a type assumption. Mappings are also abbreviated like $[d_1 \mapsto r_1, \ldots, d_n \mapsto r_n]$. The union of disjoint mappings is written as $X \cdot X'$. The restriction of $X$ to a subset $A$ of its domain is written $X \downarrow A$ and has lower precedence than union.

The types and type schemes of $\lambda\Sigma$ are defined by the following grammar:

$$\begin{array}{rcl} s \in \mathit{strengths} & = & \mathit{Int} \cup \{\infty\} \\ \alpha, \beta \in \mathit{type\_variables} & = & \text{a countably infinite set} \\ \tau \in \mathit{types} & ::= & \alpha \mid \mathit{unit} \mid \tau_1 \rightarrow \tau_2 \mid \tau\ \mathit{ref} \\ \mathit{type\_schemes} & ::= & \forall\Sigma.\tau \\ \Sigma \in \mathit{strength\_contexts} & = & \mathit{type\_variables} \xrightarrow{\mathit{fin}} \mathit{strengths} \end{array}$$

The trivial type scheme $\forall[\,].\tau$ is abbreviated by $\tau$, and type schemes that are $\alpha$-equivalent are identified. Strength contexts assign strengths to type variables.

As previously described, the semantics use two types of type assumptions. A location type assumption $\Lambda$ maps locations to types, and a variable type assumption $\Gamma$ maps variables to type schemes.

The set of free type variables of a type or type scheme is defined as usual and denoted by $FTV(\cdot)$. The sets $FTV(\Lambda)$ and $FTV(\Gamma)$ contain the free variables in the ranges of the type assumptions. This notation is extended to be $n$-ary so that, for example, $FTV(\Lambda, \Gamma) = FTV(\Lambda) \cup FTV(\Gamma)$.

As we have seen, the generalization of type variables is dependent on their criticality. To define the criticality of type variables, types and strength contexts formally, we first state that a strength is critical if it is non-positive, and non-critical otherwise. A type variable $\alpha$ is critical in $\Sigma$ if $\Sigma(\alpha)$ is critical, and a type is critical in $\Sigma$ if all of its type variables are. Similarly, a strength context is critical, written $Crit(\Sigma)$, if all of the type variables in its domain are. Non-criticality is defined similarly, but note that $\neg Crit(\Sigma)$ is not equivalent to $NonCrit(\Sigma)$.

A type judgment $\Lambda; \Gamma \vdash_\Sigma e : \tau$ reads 'With strength context $\Sigma$, given the location type assumption $\Lambda$ and variable type assumption $\Gamma$, expression $e$ has type $\tau$'. A judgment holds if it is derivable by the rules of Figure 1, which use the definitions found below.

The side conditions of the base cases ensure that a well-formedness constraint holds for all derivable judgments: the free type variables in the type assumptions and in the expression's type are in the domain of the strength context. Thus if a type variable is mentioned, its strength is explicitly given.

Instantiation is defined much as usual, but with restrictions on weak type variables. A type scheme *instantiates* to a type,

$$\vdash_\Sigma \forall[\alpha_1 \mapsto s_1, \ldots, \alpha_n \mapsto s_n].\tau \geq \tau',$$

if there exists a type substitution $S = [\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$ such that $S(\tau) = \tau'$, and for each $i$ in $\{1, \ldots, n\}$, for all $\alpha$ in $FTV(\tau_i)$, $\Sigma(\alpha) \leq s_i$. Note that by replacing $\Sigma$ with $\Sigma + 1$, which corresponds to placing a variable within one more function application context, it may become impossible to satisfy this last condition. For example, this restricts instantiation so that the following is derivable only if $s \leq 0$:

$$[\,];[\,] \vdash_{[\alpha \mapsto s]} \textit{let ref' = fn x} \Rightarrow \textit{ref x in ref' (fn z} \Rightarrow \textit{z)} : (\alpha \to \alpha)\ \textit{ref}$$

Note that the strength context is incremented in $\mathrm{APP_D}$ and decremented in $\mathrm{LAM_D}$. Addition of a constant to a strength context is defined point-wise. Infinitely strong type variables are unaffected by this, since $\infty + c = \infty$.

The side condition of $\mathrm{APP_D}$ enforces the conservative approximation described in section 2 by placing an upper bound on strengths in the strength context. The relation $Weaker(\Sigma, s)$ holds if all finite strengths in strength context $\Sigma$ are at most $s$. The similar relation where all finite and infinite strengths in $\Sigma$ are at most $s$ is used in the soundness proof and can be written $Crit(\Sigma - s)$.

The $\mathrm{REF_D}$ rule is simply a special case of $\mathrm{APP_D}$ which reflects the usual treatment of *ref* as a functional primitive having the type scheme $\forall[\alpha \mapsto 1].\alpha \to \alpha\ \textit{ref}$. In particular, its side condition ensures that reference cells have critical types.

In $\mathrm{LET_D}$ the strength contexts $\Sigma$ and $\Sigma'$ have disjoint domains by the definition of the mapping union operator. So, by the well-formedness of the second precondition,

https://doi.org/10.1017/S0956796800001593 Published online by Cambridge University Press

$$\frac{\vdash_\Sigma \Gamma(x) \succeq \tau}{\Lambda;\Gamma \vdash_\Sigma x : \tau} \quad \text{if } FTV(\Lambda,\Gamma) \text{ is a subset of } dom(\Sigma) \qquad (\text{VAR}_\text{D})$$

$$\frac{\Lambda(l) = \tau}{\Lambda;\Gamma \vdash_\Sigma l : \tau\ ref} \quad \text{if } FTV(\Lambda,\Gamma) \text{ is a subset of } dom(\Sigma) \qquad (\text{LOC}_\text{D})$$

$$\Lambda;\Gamma \vdash_\Sigma () : unit \quad \text{if } FTV(\Lambda,\Gamma) \text{ is a subset of } dom(\Sigma) \qquad (\text{UNIT}_\text{D})$$

$$\frac{\Lambda;\Gamma \vdash_\Sigma e : \tau}{\Lambda;\Gamma \vdash_\Sigma ref\ e : \tau\ ref} \quad \text{if } Crit(\Sigma \downarrow FTV(\tau)) \qquad (\text{REF}_\text{D})$$

$$\frac{\Lambda;\Gamma \vdash_\Sigma e : \tau\ ref}{\Lambda;\Gamma \vdash_\Sigma\ !e : \tau} \qquad (!_\text{D})$$

$$\frac{\Lambda;\Gamma \vdash_\Sigma e_1 : \tau\ ref \qquad \Lambda;\Gamma \vdash_\Sigma e_2 : \tau}{\Lambda;\Gamma \vdash_\Sigma e_1{:=}e_2 : unit} \qquad (:=_\text{D})$$

$$\frac{\Lambda;\Gamma \vdash_{\Sigma+1} e_1 : \tau'{\to}\tau \qquad \Lambda;\Gamma \vdash_\Sigma e_2 : \tau'}{\Lambda;\Gamma \vdash_\Sigma e_1\ e_2 : \tau} \quad \text{if } Weaker(\Sigma \downarrow FTV(\tau'), 0) \qquad (\text{APP}_\text{D})$$

$$\frac{\Lambda;\Gamma[x : \tau_1] \vdash_{\Sigma-1} e : \tau_2}{\Lambda;\Gamma \vdash_\Sigma fn\ x \Rightarrow e : \tau_1{\to}\tau_2} \quad \text{if } x \text{ is not in } dom(\Gamma) \qquad (\text{LAM}_\text{D})$$

$$\frac{\Lambda;\Gamma \vdash_{\Sigma\cdot\Sigma'} e_1 : \tau' \qquad \Lambda;\Gamma[x : \forall\Sigma'.\tau_1] \vdash_\Sigma e_2 : \tau}{\Lambda;\Gamma \vdash_\Sigma let\ x = e_1\ in\ e_2 : \tau} \qquad (\text{LET}_\text{D})$$

if $x$ is not in $dom(\Gamma)$, $NonCrit(\Sigma')$,
and $Weaker(\Sigma \downarrow (FTV(\tau') \cap dom(\Sigma)) \setminus FTV(\Lambda,\Gamma), 0)$

Fig. 1. The $\lambda\Sigma$ static semantics.

the domain of $\Sigma'$ is disjoint from the free type variables of $\Lambda$ and $\Gamma$. By using the strength context $\Sigma'$ only in the first subderivation, it is explicit that the generalized type variables are local. Furthermore, they are not critical, by the second side condition.

The last side condition of $\text{LET}_\text{D}$ states that any finite positive type variables in the type of the *let*-bound expression, but not in the type assumptions, *must* be generalized. Without this restriction, the following judgment is derivable:

$$[\,];[\,] \vdash_{[\alpha\mapsto 2]} let\ x = fn\ z \Rightarrow fn\ y \Rightarrow ref\ y\ in\ x() : \alpha{\to}\alpha\ ref.$$

This expression should only be typable with $\alpha$ having a strength of 1 or less, like the observationally equivalent $fn\ y \Rightarrow ref\ y$, since applying it to one argument creates a cell. The problem here is that if $\alpha$ is not generalized, then its strength might not be properly decremented in the *let* body, as the appropriate decrementing is to be enforced by instantiation.

This is a very technical condition. It disallows the rule's use if any type variable in $(FTV(\tau') \cap dom(\Sigma)) \setminus FTV(\Lambda,\Gamma)$ is finite and positive in $\Sigma$. But if this is the case,

https://doi.org/10.1017/S0956796800001593 Published online by Cambridge University Press

there exists another derivation which types the *let* expression by renaming these type variables with fresh ones and adding them to $\Sigma'$.

## 4 Soundness

This section defines the dynamic semantics of $\lambda\Sigma$ and outlines its proof of soundness of typability.

The dynamic semantics defines to what answer, if any, an expression evaluates. An answer is either the symbol *wrong* or a value, which is a closed expression defined by the following grammar:

$$\begin{array}{rcl} v \in \mathit{values} & ::= & l \mid () \mid \mathit{fn}\ x \Rightarrow e \\ a \in \mathit{answers} & ::= & v \mid \mathit{wrong} \end{array}$$

Since values are expressions, those that are $\alpha$-equivalent are identified.

We use the the standard rules shown in Figure 2 to define the semantics in terms of judgments of the form $\sigma \vdash e \Longrightarrow a, \sigma'$, which reads 'Given the store $\sigma$, the expression $e$ evaluates to answer $a$, resulting in a new store $\sigma'$'. A store $\sigma$ is a finite mapping from locations to values.

To relate the mappings of locations used in the dynamic and static semantics, we define that a store $\sigma$ *type-matches* a location type assumption $\Lambda$ with respect to $\Sigma$, written $\vdash_\Sigma \sigma : \Lambda$, if $\mathit{dom}(\sigma) = \mathit{dom}(\Lambda)$, and $\Lambda; [\,] \vdash_\Sigma \sigma(l) : \Lambda(l)\ \mathit{ref}$ for all $l$ in $\mathit{dom}(\sigma)$.

Soundness is shown via a form of type preservation under evaluation (Harper, 1993; Hindley and Seldin, 1986; Tofte, 1988; Wright and Felleisen, 1991). Unfortunately, while evaluation preserves types, it does not necessarily preserve strengths. For example, consider the following expressions and value:

$$\begin{array}{rcl} e_0 & = & \mathit{fn}\ z \Rightarrow z \\ e_1 & = & \mathit{fn}\ x \Rightarrow \mathit{fn}\ y \Rightarrow x\ e_0\ e_0 \\ e_2 & = & \mathit{fn}\ a \Rightarrow (\mathit{let}\ b = \mathit{ref}\ a\ \mathit{in}\ \mathit{fn}\ c \Rightarrow\ !b)() \\ e & = & e_1\ e_2 \\ v & = & \mathit{fn}\ y \Rightarrow e_2\ e_0\ e_0. \end{array}$$

The expression $e_2$ is a function that is assigned a critical type, i.e. the judgment

$$[\,]; [\,] \vdash_{[\alpha \mapsto s]} e_2 : ((\alpha\to\alpha)\to(\alpha\to\alpha))\to((\alpha\to\alpha)\to(\alpha\to\alpha))$$

is derivable only if $s \leq 0$. The expression $e$ evaluates to $v$, and the strongest typings for these are

$$[\,]; [\,] \vdash_{[\alpha \mapsto 0, \beta \mapsto \infty]} e : \beta\to\alpha\to\alpha$$

$$[\,]; [\,] \vdash_{[\alpha \mapsto -1, \beta \mapsto \infty]} v : \beta\to\alpha\to\alpha.$$

In $\lambda\Sigma$, $v$ cannot be typed with the higher strengths of $e$'s type derivation, even though that would be sound. However, note that only the strength of $\alpha$ is not preserved under evaluation, and it is critical in both judgments. The type preservation theorem will show that all non-critical strengths in an expression's type are preserved under evaluation.

$$\sigma \vdash v \Longrightarrow v, \sigma \qquad \text{(VAL)}$$

$$\frac{\sigma \vdash e \Longrightarrow v, \sigma'}{\sigma \vdash \mathit{ref}\ e \Longrightarrow l, \sigma'[l \mapsto v]} \ \text{if } l \text{ is not in } \mathit{dom}(\sigma') \qquad \text{(ALLOC)}$$

$$\frac{\sigma \vdash e \Longrightarrow l, \sigma'}{\sigma \vdash !e \Longrightarrow \sigma'(l), \sigma'} \qquad \text{(CONT)}$$

$$\frac{\sigma \vdash e_1 \Longrightarrow l, \sigma_1 \qquad \sigma_1 \vdash e_2 \Longrightarrow v, \sigma_2[l \mapsto v']}{\sigma \vdash e_1 {:=} e_2 \Longrightarrow (), \sigma_2[l \mapsto v]} \qquad \text{(UPD)}$$

$$\frac{\sigma \vdash e_1 \Longrightarrow \mathit{fn}\ x \Rightarrow e_1', \sigma_1 \qquad \sigma_1 \vdash e_2 \Longrightarrow v_2, \sigma_2 \qquad \sigma_2 \vdash [v_2/x]e_1' \Longrightarrow v, \sigma'}{\sigma \vdash e_1\ e_2 \Longrightarrow v, \sigma'} \qquad \text{(APPLY)}$$

$$\frac{\sigma \vdash e_1 \Longrightarrow v_1, \sigma_1 \qquad \sigma_1 \vdash [v_1/x]e_2 \Longrightarrow v_2, \sigma_2}{\sigma \vdash \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \Longrightarrow v_2, \sigma_2} \qquad \text{(BIND)}$$

$$\frac{\sigma \vdash e \Longrightarrow \mathit{wrong}, []}{\sigma \vdash \mathit{ref}\ e \Longrightarrow \mathit{wrong}, []} \qquad \text{(ALLOC-}\mathit{wrong}\text{)}$$

$$\frac{\sigma \vdash e \Longrightarrow a, \sigma'}{\sigma \vdash !e \Longrightarrow \mathit{wrong}, []} \ \text{if } a \text{ is not a location} \qquad \text{(CONT-}\mathit{wrong}\text{)}$$

$$\frac{\sigma \vdash e_1 \Longrightarrow a_1, \sigma_1 \qquad \sigma_1 \vdash e_2 \Longrightarrow a_2, \sigma_2}{\sigma \vdash e_1 {:=} e_2 \Longrightarrow \mathit{wrong}, []} \ \begin{array}{l}\text{if } a_1 \text{ is not a location,}\\ a_2 = \mathit{wrong},\\ \text{or } l \text{ is not in } \mathit{dom}(\sigma_1)\end{array} \qquad \text{(UPD-}\mathit{wrong}\text{)}$$

$$\frac{\sigma \vdash e_1 \Longrightarrow a_1, \sigma_1 \qquad \sigma_1 \vdash e_2 \Longrightarrow a_2, \sigma_2 \qquad \sigma_2 \vdash [a_2/x]e_1' \Longrightarrow a, \sigma'}{\sigma \vdash e_1\ e_2 \Longrightarrow \mathit{wrong}, []} \ \begin{array}{l}\text{if } a_1 \text{ is not a function,}\\ a_2 = \mathit{wrong},\\ \text{or } a = \mathit{wrong}\end{array} \qquad \text{(APPLY-}\mathit{wrong}\text{)}$$

$$\frac{\sigma \vdash e_1 \Longrightarrow a_1, \sigma_1 \qquad \sigma_1 \vdash [a_1/x]e_2 \Longrightarrow a_2, \sigma_2}{\sigma \vdash \mathit{let}\ x = e_1\ \mathit{in}\ e_2 \Longrightarrow \mathit{wrong}, []} \ \begin{array}{l}\text{if } a_1 = \mathit{wrong},\\ \text{or } a_2 = \mathit{wrong}\end{array} \qquad \text{(BIND-}\mathit{wrong}\text{)}$$

Fig. 2. The dynamic semantics.

In most cases, including whenever critical strengths are not involved, $\lambda\Sigma$ types the value with the same or higher strengths than the expression. For a simple example, consider the following expression and its value:

$$e = (\mathit{fn}\ x \Rightarrow x)(\mathit{fn}\ y \Rightarrow \mathit{ref}\ y)$$
$$v = \mathit{fn}\ y \Rightarrow \mathit{ref}\ y.$$

Type preservation will show that since $[\,];[\,] \vdash_{[\alpha \mapsto 0]} e : \alpha \rightarrow \alpha\ \mathit{ref}$, then there exists a critical strength $s$ such that $[\,];[\,] \vdash_{[\alpha \mapsto s]} v : \alpha \rightarrow \alpha\ \mathit{ref}$. But $v$ is also typable with the strength context $[\alpha \mapsto 1]$.

To formalize this idea of weakening strengths, we define a relation on strength contexts. A strength context $\Sigma'$ is *weaker (below s)* than strength context $\Sigma$ of the same domain, $\Sigma' \leq_s \Sigma$, if for all $\alpha$ in $\mathit{dom}(\Sigma')$, $\Sigma'(\alpha) = \Sigma(\alpha)$ whenever $s \leq \Sigma(\alpha)$, and

https://doi.org/10.1017/S0956796800001593 Published online by Cambridge University Press

$\Sigma'(\alpha) \le \Sigma(\alpha)$ otherwise. Of particular interest is the case when $s = 1$, as then the two strength contexts agree on all positive strengths, and $\Sigma'$ is said to be *more critical* than $\Sigma$. Note that $\Sigma' + 1 \le_s \Sigma + 1$ implies $\Sigma' \le_s \Sigma$, which implies $\Sigma' \le_{s+1} \Sigma$, but the converses do not hold.

### *Soundness proof outline*

The remainder of this section presents an overview of the soundness proof. This presentation is in a top-down manner, giving the theorems first in order to motivate the needed lemmas. For the full proof, refer to the Appendix, which gives the theorems and lemmas in the order of their dependence. This proof is relatively complex because of the need for tight control over the strengths of type variables.

The soundness of the static semantics is the conjunction of the following two theorems. The first states that evaluation preserves types, i.e. if $e$ evaluates to $v$, then $e$ and $v$ have the same type. As previously explained, $v$ may require more critical strengths. Furthermore, any cells created during this evaluation have critical types. The second theorem states that any well-typed expression does not 'go wrong', i.e. it either evaluates to a value, or it diverges.

**Theorem (Top-level type preservation under evaluation)**
*If* $[\,] \vdash e \Longrightarrow v, \sigma'$, *and* $[\,];[\,] \vdash_\Sigma e : \tau$, *then there exist* $\Lambda_0$ *and* $\Sigma_0$ *such that*

1. $\Sigma_0 \le_1 \Sigma$,
2. $\Lambda_0;[\,] \vdash_{\Sigma_0} v : \tau$,
3. $\vdash_{\Sigma_0} \sigma' : \Lambda_0$, *and*
4. $Crit\,(\Sigma_0 \downarrow FTV(\Lambda_0))$.

**Proof Sketch** This is proved by generalizing the theorem to all memories and location type assumptions, generalizing the form of some of the strength contexts, and then using structural induction on the evaluation derivation. The APPLY and BIND cases require the Value Substitution lemma below that describes the effect on the typability of an expression when substituting a value for a variable in that expression. The last hypothesis of that lemma is achieved through the side conditions on the $\mathrm{APP_D}$ and $\mathrm{LET_D}$ rules.

Most of the cases make extensive use of weakening and strengthening lemmas. Weakening describes when hypotheses can be safely added and when strengths can be decreased in a typing judgment. In particular, $\lambda\Sigma$ does *not* allow infinite strengths to be decreased arbitrarily. For example,

$$[\,];[\,] \vdash_{[\alpha \mapsto s, \beta \mapsto \infty]} fn\, f \Rightarrow fn\ x \Rightarrow f\ x : (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$$

is derivable only if $s = \infty$, or $s \le 2$. Conversely, Strengthening and Strength Context Strengthening describe when unused hypotheses can be removed from a typing judgment. □

To relate this theorem to the preceding example of the lack of strength preservation, define $\Sigma = [\alpha \mapsto 0, \beta \mapsto \infty]$, and $\Sigma_0 = [\alpha \mapsto -1, \beta \mapsto \infty]$. No cells are created by that evaluation, so $\sigma' = [\,]$, and $\lambda_0 = [\,]$.

**Theorem (Well-typed programs do not go wrong)**
*If* $[\,];[\,] \vdash_\Sigma e : \tau$, *then* $[\,] \not\vdash e \Longrightarrow wrong, [\,]$.

**Proof Sketch** Like the previous theorem, this is proved by generalizing the judgments to all memories and location type assumptions and generalizing the form of the strength contexts. Then we assume that $e$ *does* go wrong and use structural induction on the evaluation derivation to show that it cannot be typed, essentially the contrapositive of the above statement.

If $e$ is a value, then it cannot go wrong. Otherwise, there are two classes of cases for $e$ to go wrong. First, if one of the hypotheses of the evaluation relation goes wrong, then induction shows that that expression cannot be typed. Second, one of the hypotheses of the evaluation relation could result in a value of the wrong form, e.g. if $e = !e'$, and $e'$ evaluates to an abstraction. Using proof by contradiction and assuming that $e$ is typable, the Type Preservation theorem shows in the first case that each of the expressions in the hypothesis of the evaluation rule are typable, and in the second case, that any intermediate values are of the expected form. Since both cases lead to contradictions, $e$ is not typable.

For the APPLY and BIND cases, the following Value Substitution lemma shows the typability of the needed substitution instances. □

The Value Substitution lemma states that, under restrictions, the type of an expression is stable under substitution of a value for a variable of more general type. The result of the substitution may require more critical strengths.

**Lemma (Value substitution)**
*If* 1. $\Lambda;\Gamma[x : \forall\Sigma_2.\tau_2] \vdash_\Sigma e : \tau_1$,
2. $\Lambda;[\,] \vdash_{\Sigma\cdot\Sigma_2} v : \tau_2$, *and*
3. $Weaker(\Sigma \downarrow FTV(\Lambda,\tau_2) \cap dom(\Sigma), 0)$,

*then there exists a* $\Sigma'$ *such that* $\Sigma' \leq_1 \Sigma$, *and* $\Lambda;\Gamma \vdash_{\Sigma'} [v/x]e : \tau_1$.

**Proof Sketch** We generalize the statement of this lemma to prove it by structural induction on the type derivation. In general, the strength context for the first hypothesis is of the form $(\Sigma \cdot \Sigma_1) + c$, and for the conclusion, $(\Sigma' \cdot \Sigma_1) + c$. The strength context $\Sigma_1$ accounts for the local type variables in the $\mathrm{LET_D}$ case, which cannot be allowed to decrease. The constant $c$ generalizes the constant 1 or $-1$ added in the $\mathrm{APP_D}$ and $\mathrm{LAM_D}$ cases. With this generalization, the only interesting case is when $e = x$, which is proved as an instance of the following Type Substitution lemma and Weakening. □

The basic idea of the Type Substitution lemma is to show that the type of an expression is stable when substituting types for its type variables. The strength context $\Sigma_1$ contains those type variables being substituted by $S$, whereas $\Sigma_2$ contains those added by the substitution. Critical strengths of type variables unaffected by the substitution may need to be lowered. The constant $c$ and strength context $\Sigma_1$ added by the generalization of the Value Substitution lemma carry over to this lemma.

**Lemma (Type substitution)**

*If* 1. $\Lambda;\Gamma \vdash_{\Sigma\cdot\Sigma_1} e : \tau$,
2. $S = [\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$,
3. $dom(S) = dom(\Sigma_1)$,
4. *for all* $i$ *in* $\{1,\ldots,n\}$, $Crit((\Sigma \cdot \Sigma_2) + c - \Sigma_1(\alpha_i) \downarrow FTV(\tau_i))$, *and*
5. $Weaker(\Sigma \downarrow FTV(\Lambda,\Gamma,\tau) \cap dom(\Sigma), 0)$,

*then there exists a* $\Sigma'$ *such that* $\Sigma' \leq_1 \Sigma$, *and* $S(\Lambda); S(\Gamma) \vdash_{(\Sigma'\cdot\Sigma_2)+c} e : S(\tau)$.

**Proof Sketch** As with Value Substitution, the strength contexts are generalized so that the lemma may be proved by structural induction on the type derivation. In particular, the first hypothesis uses $(\Sigma \cdot \Sigma_1 \cdot \Sigma_0) + c'$, and the conclusion uses $(((\Sigma' \cdot \Sigma_2) + c) \cdot \Sigma_0) + c'$. The strength context $\Sigma_0$ accounts for the local variables in $\mathrm{LET_D}$, and the constant $c'$ accounts for the constants used in $\mathrm{APP_D}$ and $\mathrm{LAM_D}$.

The $\mathrm{VAR_D}$ case is then proved by finding an appropriate $\Sigma'$ such that the required instantiation holds. Similarly, the $\mathrm{REF_D}$, $\mathrm{APP_D}$, and $\mathrm{LET_D}$ cases require calculating an appropriate $\Sigma'$ such that the side conditions hold, as well as using a Ground Type Substitution lemma to eliminate some type variables from consideration. The remaining cases follow by induction alone. □

## 5 Algorithmic formalisms

There are other ways to formalize weak type polymorphism. For example, Hoang *et al.* (1993) describe an alternative, which we compare to $\lambda\Sigma$ in section 7. Furthermore, the implementation of SML/NJ does not resemble the formalism of $\lambda\Sigma$ directly. This section discusses how to adapt the declarative formalism to a more algorithmic approach similar to that of SML/NJ.

We introduce the concepts in this SML/NJ-like formalism in three stages, using three pairs of formalisms, each consisting of a declarative and an algorithmic formalism. First, we define the equivalent formalisms $\lambda\Sigma^-$ and $\lambda\Psi^-$ to relate the two families of formalisms and to motivate the algorithmic framework. Next, we improve the application rules of $\lambda\Psi^-$ to obtain $\lambda\Psi$ and show its similarity to $\lambda\Sigma$. Finally, we improve application rules and add a weakening rule to obtain $\lambda\Sigma^+$ and $\lambda\Psi^+$. This last formalism is the most like that of SML/NJ given in this paper. Of these additional formalisms, only $\lambda\Sigma^+$ is known to be sound.

### *Type systems $\lambda\Sigma^-$ and $\lambda\Psi^-$*

A primary idea of the algorithmic approach is to explicitly relate the type and strengths of an expression to a syntactic context. Since this context can be arbitrarily deep, an approximation, called an *occurrence*, is defined. For this paper, we consider only two components of SML/NJ's occurrences: the *abstraction depth* and the *maximum strength.*

Consider the typing rules as an algorithm. Incrementing and decrementing the

| | | |
|---|---|---|
| $\dfrac{\Lambda;\Gamma \vdash_\Sigma e : \tau}{\Lambda;\Gamma \vdash_\Sigma \mathit{ref}\ e : \tau\ \mathit{ref}}$ if $\mathit{Crit}(\Sigma \downarrow \mathit{FTV}(\tau))$, and $\mathit{Weaker}(\Sigma \downarrow \mathit{FTV}(\Lambda,\Gamma),0)$ | | $(\mathrm{REF}_\mathrm{D}^-)$ |
| $\dfrac{\Lambda;\Gamma \vdash_{\Sigma+1} e_1 : \tau'\rightarrow\tau \qquad \Lambda;\Gamma \vdash_\Sigma e_2 : \tau'}{\Lambda;\Gamma \vdash_\Sigma e_1\ e_2 : \tau}$ if $\mathit{Weaker}(\Sigma \downarrow \mathit{FTV}(\Lambda,\Gamma,\tau'),0)$ | | $(\mathrm{APP}_\mathrm{D}^-)$ |

Fig. 3. Changes to $\lambda\Sigma$ that results in the $\lambda\Sigma^-$ static semantics.

strengths of all type variables in the strength context at every abstraction and application is inefficient, and it would be better to use a single offset, the abstraction depth, instead. We can split the strength context $\Sigma$ into a strength context $\Psi$ and an abstraction depth $d$ such that $\Sigma = \Psi - d$, and the typing rules can be written such that $\Psi$ is fixed. Then the negation of the abstraction depth of a given subexpression is a lower bound on the number of application contexts within which the subexpression is a function.

To motivate the maximum strength, we digress temporarily. Using the more restrictive application typing rules given in Figure 3 results in the $\lambda\Sigma^-$ type system. These stronger side conditions will simplify the transition to a more algorithmic formalism.

However, the Value Substitution lemma surprisingly does *not* hold in $\lambda\Sigma^-$. For example, consider the following expression and value:

$$\begin{aligned} v &= \mathit{fn}\ a \Rightarrow a()()() \\ e &= \mathit{fn}\ y \Rightarrow \mathit{let}\ u = \mathit{ref}\ y\ \mathit{in}\ x, \end{aligned}$$

which are assigned non-critical types:

$$[\,];[\,] \vdash_{[\alpha\mapsto\infty]} v : (\mathit{unit}\rightarrow\mathit{unit}\rightarrow\mathit{unit}\rightarrow\alpha)\rightarrow\alpha$$

$$[\,];[x : (\mathit{unit}\rightarrow\mathit{unit}\rightarrow\mathit{unit}\rightarrow\alpha)\rightarrow\alpha] \vdash_{[\alpha\mapsto\infty,\beta\mapsto 1]} e : \beta\rightarrow(\mathit{unit}\rightarrow\mathit{unit}\rightarrow\mathit{unit}\rightarrow\alpha)\rightarrow\alpha.$$

But the best typing of their appropriate substitution is

$$[\,];[\,] \vdash_{[\alpha\mapsto\infty,\beta\mapsto 0]} [v/x]e : \beta\rightarrow(\mathit{unit}\rightarrow\mathit{unit}\rightarrow\mathit{unit}\rightarrow\alpha)\rightarrow\alpha,$$

where the non-critical strength of $\beta$ has been lowered. As a result, type preservation in $\lambda\Sigma^-$ is still open. But well-typed expressions do not go wrong, since anything well-typed in $\lambda\Sigma^-$ is also well-typed in $\lambda\Sigma$.

We can now explain the maximum strength $m$, which is an upper bound on the finite strengths in $\Psi$. Its use allows the side condition on $\mathrm{APP}_\mathrm{D}^-$ to be replaced by a side condition on $\mathrm{VAR}_\mathrm{A}$. There is no reason to prefer the latter in the context of $\lambda\Sigma^-$, but some motivation for this change will be given within the context of $\lambda\Sigma$.

The 'top-level' occurrence is named *Root*, and the functions on occurrences are named *Rator*$(\cdot)$, *Rand*$(\cdot)$, *Abs*$(\cdot)$, and *Let*$(\cdot)$, corresponding to the possible syntactic contexts of application functions, application arguments, abstraction bodies and *let*-bound expressions, respectively. The function corresponding to the syntactic context of a *let* body is the identity. These functions on occurrences $(d, m)$ are then

$$\frac{\vdash_{\Psi-d} \Gamma(x) \geq \tau}{\Lambda;\Gamma \vdash_{\Psi;d,m} x : \tau} \text{ if } Weaker(\Psi \downarrow FTV(\Lambda,\Gamma,\tau), m) \qquad (\text{VAR}_\text{A})$$

$$\frac{\Lambda(l) = \tau}{\Lambda;\Gamma \vdash_{\Psi;d,m} l : \tau\ ref} \text{ if } Weaker(\Psi \downarrow FTV(\Lambda,\Gamma), m) \qquad (\text{LOC}_\text{A})$$

$$\Lambda;\Gamma \vdash_{\Psi;d,m} () : unit \text{ if } Weaker(\Psi \downarrow FTV(\Lambda,\Gamma), m) \qquad (\text{UNIT}_\text{A})$$

$$\frac{\Lambda;\Gamma \vdash_{\Psi;Rand(d,m)} e : \tau}{\Lambda;\Gamma \vdash_{\Psi;d,m} ref\ e : \tau\ ref} \text{ if } Crit(\Psi - d \downarrow FTV(\tau)) \qquad (\text{REF}_\text{A})$$

$$\frac{\Lambda;\Gamma \vdash_{\Psi;d,m} e : \tau\ ref}{\Lambda;\Gamma \vdash_{\Psi;d,m} !e : \tau} \qquad (!_\text{A})$$

$$\frac{\Lambda;\Gamma \vdash_{\Psi;d,m} e_1 : \tau\ ref \qquad \Lambda;\Gamma \vdash_{\Psi;d,m} e_2 : \tau}{\Lambda;\Gamma \vdash_{\Psi;d,m} e_1 := e_2 : unit} \qquad (:=_\text{A})$$

$$\frac{\Lambda;\Gamma \vdash_{\Psi;Rator(d,m)} e_1 : \tau' \rightarrow \tau \qquad \Lambda;\Gamma \vdash_{\Psi;Rand(d,m)} e_2 : \tau'}{\Lambda;\Gamma \vdash_{\Psi;d,m} e_1\ e_2 : \tau} \qquad (\text{APP}_\text{A})$$

$$\frac{\Lambda;\Gamma[x : \tau_1] \vdash_{\Psi;Abs(d,m)} e : \tau_2}{\Lambda;\Gamma \vdash_{\Psi;d,m} fn\ x \Rightarrow e : \tau_1 \rightarrow \tau_2} \text{ if } x \text{ is not in } dom(\Gamma) \qquad (\text{LAM}_\text{A})$$

$$\frac{\Lambda;\Gamma \vdash_{\Psi \cdot \Psi';Let(d,m)} e_1 : \tau' \qquad \Lambda;\Gamma[x : \forall\Psi' - d.\tau_1] \vdash_{\Psi;d,m} e_2 : \tau}{\Lambda;\Gamma \vdash_{\Psi;d,m} let\ x = e_1\ in\ e_2 : \tau} \qquad (\text{LET}_\text{A})$$

if $x$ is not in $dom(\Gamma)$, $NonCrit(\Psi' - d)$,
and $Weaker(\Psi \downarrow (FTV(\tau') \cap dom(\Psi)) \setminus FTV(\Lambda,\Gamma), d)$

Fig. 4. The $\lambda\Psi^-$ static semantics.

defined by

$$\begin{aligned} Rand(d,m) &= (d, \min(d,m)) \\ Rator(d,m) &= (d-1, m) \\ Abs(d,m) &= (d+1, m) \\ Let(d,m) &= (d, \infty) \\ Root &= (0, \infty) \end{aligned}$$

A type judgment $\Lambda;\Gamma \vdash_{\Psi;d,m} e : \tau$ reads 'With strength context $\Psi$, at the occurrence containing the abstraction depth $d$ and maximum strength $m$, and given the location type assumption $\Lambda$ and variable type assumption $\Gamma$, expression $e$ has type $\tau$'. Derivability in $\lambda\Psi^-$ is defined by the rules of Figure 4.

As in $\lambda\Sigma$, the base cases have a side condition which ensures that all derivable judgments are well-formed. Only the abstraction depth is incremented in $\text{APP}_\text{A}$ and decremented in $\text{LAM}_\text{A}$. But $\text{VAR}_\text{A}$, $\text{REF}_\text{A}$ and $\text{LET}_\text{A}$ use the strength context offset by the abstraction depth. In $\text{LET}_\text{A}$ the second precondition ensures that

https://doi.org/10.1017/S0956796800001593 Published online by Cambridge University Press

| | | |
|---|---|---|
| $\dfrac{\vdash_{\Psi\text{-}d} \Gamma(x) \succeq \tau}{\Lambda;\Gamma \vdash_{\Psi;d,m} x : \tau}$ | if $FTV(\Lambda,\Gamma)$ is a subset of $dom(\Psi)$, and $Weaker(\Psi \downarrow FTV(\tau), m)$ | ($\text{VAR}_\text{A}$) |
| $\dfrac{\Lambda(l) = \tau}{\Lambda;\Gamma \vdash_{\Psi;d,m} l : \tau\ ref}$ | if $FTV(\Lambda,\Gamma)$ is a subset of $dom(\Psi)$ | ($\text{LOC}_\text{A}$) |
| $\Lambda;\Gamma \vdash_{\Psi;d,m} () : unit$ | if $FTV(\Lambda,\Gamma)$ is a subset of $dom(\Psi)$ | ($\text{UNIT}_\text{A}$) |

Fig. 5. Changes to $\lambda\Psi^-$ that result in the $\lambda\Psi$ static semantics.

$m$ is an upper bound on the strengths in $\Psi$. Since $Let(d,m) = (d,\infty)$, the first precondition enforces only a trivial upper bound on the strengths in $\Psi'$. If instead $Let(d,m) = (d,m)$, then there would be no generalization in expressions such as $f\ (let\ x = fn\ z \Rightarrow ref\ z\ in\ e)$. Similarly, the top-level occurrence *Root* also has an infinite maximum strength, so that the empty syntactic context places no upper bound on strengths.

Now we show that the two systems are equivalent. First, the side conditions on the application cases in $\lambda\Sigma^-$ are satisfied via the side conditions on the base cases in $\lambda\Psi^-$, as shown by the following lemma.

**Lemma (Maximum strength for $\lambda\Psi^-$)**
*If* $\Lambda;\Gamma \vdash_{\Psi;d,m} e : \tau$, *then* $Weaker(\Psi \downarrow FTV(\Lambda,\Gamma,\tau), m)$.

This lemma and the following two each hold by structural induction. Using that lemma, we can prove that anything typable in $\lambda\Psi^-$ is typable in $\lambda\Sigma^-$ with the same type and the strengths offset by the abstraction depth:

**Lemma ($\lambda\Psi^-$ typability implies $\lambda\Sigma^-$ typability)**
*If* $\Lambda;\Gamma \vdash_{\Psi;d,m} e : \tau$, *then* $\Lambda;\Gamma \vdash_{\Psi-d} e : \tau$.

The converse relationship holds if $m$ bounds the appropriate strengths.

**Lemma ($\lambda\Sigma^-$ typability implies $\lambda\Psi^-$ typability)**
*If* $\Lambda;\Gamma \vdash_{\Psi-d} e : \tau$, *and* $Weaker(\Psi \downarrow FTV(\Lambda,\Gamma,\tau), m)$, *then* $\Lambda;\Gamma \vdash_{\Psi;d,m} e : \tau$.

In particular, this means that the two semantics admit corresponding type derivations at the top-level, where $m = \infty$, and thus the second hypothesis holds trivially.

## *Type system $\lambda\Psi$*

The SML/NJ implementation is not as restrictive as $\lambda\Psi^-$ in its use of the maximum strength. Replacing the first three rules in Figure 4 with the corresponding rules in Figure 5 results in a system $\lambda\Psi$ that is more like $\lambda\Sigma$ and the implementation.

These changes seem natural, e.g. the side condition of $\text{VAR}_\text{A}$ does not place constraints on type variables not in the type of the variable as does $\text{VAR}_\text{A}^-$. When comparing this sytem to $\lambda\Sigma$ and considering the typing rules as an algorithm, it can be argued that the side condition of $\text{VAR}_\text{A}$ is more efficient than that of $\text{APP}_\text{D}$ since instantiation must examine the strengths of some of the type variables of $\tau$ anyway.

$$\frac{\Lambda;\Gamma \vdash_{\Sigma+1} e_1 : \tau' \rightarrow \tau \qquad \Lambda;\Gamma \vdash_{\Sigma'} e_2 : \tau'}{\Lambda;\Gamma \vdash_{\Sigma} e_1\ e_2 : \tau} \qquad (\text{APP}^{+}_{\text{D}})$$

if $Weaker(\Sigma' \downarrow FTV(\tau'), 0)$,
and $\Sigma'(\alpha) \geq \Sigma(\alpha)$ if $\alpha$ is in $FTV(\tau') \setminus FTV(\Lambda, \Gamma)$,
$\Sigma'(\alpha) = \Sigma(\alpha)$ otherwise

$$\frac{\Lambda;\Gamma \vdash_{\Sigma} e : \tau}{\Lambda;\Gamma \vdash_{\Sigma'} e : \tau} \quad \text{if } \Sigma' \text{ is point-wise less than or equal to } \Sigma \qquad (\text{WEAKEN}_{\text{D}})$$

Fig. 6. Changes to $\lambda\Sigma$ that result in the $\lambda\Sigma^{+}$ static semantics.

This system is strictly less conservative than $\lambda\Sigma^{-}$ and $\lambda\Psi^{-}$, but is incomparable with $\lambda\Sigma$. For example, if

$$e = (fn\ z \Rightarrow z)(fn\ a \Rightarrow let\ x = fn\ y \Rightarrow ref\ a\ in\ fn\ b \Rightarrow ref\ b),$$

then in $\lambda\Psi$ we have $[\,];[\,] \vdash_{[\alpha \mapsto 2, \beta \mapsto 0];Root} e : \alpha \rightarrow \beta \rightarrow \beta\ ref$. But in $\lambda\Sigma$, the finite strengths of the argument's type must be critical, so $\alpha$ is at most 0. And if

$$e = (fn\ z \Rightarrow (fn\ x \Rightarrow fn\ y \Rightarrow z)(z := fn\ a \Rightarrow ref\ a)())(ref\ (fn\ a \Rightarrow ref\ a)),$$

then in $\lambda\Sigma$ we have $[\,];[\,] \vdash_{[\alpha \mapsto 0]} e : \alpha \rightarrow \alpha\ ref$. But in $\lambda\Psi$, $\alpha$ has strength at most $-1$ at the top-level occurrence. It is open whether $\lambda\Psi$ is sound or even stable under substitution.

### *Type systems $\lambda\Sigma^{+}$ and $\lambda\Psi^{+}$*

As previously discussed, when typing an application expression any finite strengths of the argument's type must be critical because the function may, in turn, apply the argument to other arguments, possibly creating a reference cell. However, if the argument is purely functional, no cell can be created that is not already reflected by the type and strengths of the function. In $\lambda\Sigma$, this results in overly conservative strengths as in

$$[\,];[\,] \vdash_{[\alpha \mapsto 0, \beta \mapsto \infty]} (fn\ x \Rightarrow fn\ y \Rightarrow ref\ x)(fn\ a \Rightarrow a) : \beta \rightarrow (\alpha \rightarrow \alpha\ ref).$$

The argument, the identity function, is given a weak type to match the strength of the function domain.

The application rule of Figure 6 does not force the conservative approximation on the type variables of the argument which 'could have been' of infinite strength. The system $\lambda\Sigma^{+}$, which replaces the application rule of $\lambda\Sigma$ with this one, is strictly less conservative than $\lambda\Sigma$, and the previous example can be typed with strength context $[\alpha \mapsto 1, \beta \mapsto \infty]$.

The soundness proof extends to this system with little modification once a weakening rule such as $\text{WEAKEN}_{\text{D}}$ is also added. This rule allows infinite strengths to be weakened to any finite strength, as in SML/NJ, and unlike the Weakening lemma. Because of Weakening, it would be sufficient for the side condition to state that $\Sigma$ and $\Sigma'$ agree on all finite values in $\Sigma$. Without such a rule, the system is not

$$\frac{\Lambda;\Gamma \vdash_\Sigma e_1 : \tau' \qquad \Lambda;\Gamma \vdash_\Sigma e_2 : \tau}{\Lambda;\Gamma \vdash_\Sigma e_1 ; e_2 : \tau} \quad (\mathrm{SEQ_D})$$

Fig. 7. Sequencing typing rule.

stable under substitution. This weakening rule is only useful immediately preceding application, so the typing rules could be combined, which would restore the syntax-directedness of the system, but would further complicate the side conditions on the application rule.

The algorithmic system $\lambda\Psi^+$ can be defined similarly, and is the most SML/NJ-like system in this paper. Like $\lambda\Psi$, its soundness is still open.

## 6 Relation to SML/NJ

None of these formalisms completely models the implementation, and this section describes some of the differences. Syntactic differences include that SML/NJ types correspond to the pairing of types and strength contexts and that the formalisms restrict the reference primitives to their fully applied forms. It is almost equivalent to replace the given inference rules for the reference primitives with the following:

$$\begin{aligned} \mathit{ref} &: \forall[\alpha \mapsto 1].\alpha \to \alpha\ \mathit{ref} \\ ! &: \forall[\alpha \mapsto \infty].\alpha\ \mathit{ref} \to \alpha \\ := &: \forall[\alpha \mapsto \infty].\alpha\ \mathit{ref} \to \alpha \to \mathit{unit} \end{aligned}$$

in the variable type assumption, or as the equivalent axioms. A disadvantage of this alternative is that values must be given to the evaluation of partially applied primitives, which complicates the dynamic semantics and loosens the correspondence of the inference rules of the static and dynamic semantics. Furthermore, use of ! and := would then involve the conservative approximation of general application.

Sequencing, $e_1;e_2$, can be treated as syntactic sugar for *let* $z = e_1$ *in* $e_2$, where $z$ is new. Or a type inference rule such as that in Figure 7 can be added. Either option admits the same expressions, although the definition as a *let* expression allows more derivations. The other traditional definition, that of the semicolon (;) as an infix function, unnecessarily involves the conservative approximation of application.

The implementation has additional fields in the occurrence to more accurately approximate the syntactic contexts. For example, one field allows curried function applications to be treated somewhat like a single uncurried application, by using the same occurrence in typing each of its arguments. This corresponds to having a single application rule for typing multiple curried arguments at once, as in Figure 8.

## 7 Comparisons with other related systems

Weak polymorphism is usually explained as a generalization of Tofte's imperative type system, but this is not entirely correct. Tofte's system uses *two* inference rules

$$\frac{\Lambda;\Gamma \vdash_{\Sigma+n} e_0 : \tau_1 \rightarrow \cdots \tau_n \rightarrow \tau \qquad \Lambda;\Gamma \vdash_{\Sigma} e_i : \tau_i \quad \text{for all } i \text{ in } \{1,\ldots,n\}}{\Lambda;\Gamma \vdash_{\Sigma} e_0\ e_1 \ldots e_n : \tau} \quad (\mathrm{APPmany}_{\mathrm{D}}^{+})$$

$$\begin{array}{l} \text{if } \mathit{Weaker}(\Sigma' \downarrow \mathit{FTV}(\tau_1,\ldots,\tau_n), 0), \\ \text{and } \Sigma'(\alpha) \geq \Sigma(\alpha) \text{ if } \alpha \text{ is in } \mathit{FTV}(\tau_1,\ldots,\tau_n) \setminus \mathit{FTV}(\Lambda,\Gamma), \\ \quad \Sigma'(\alpha) = \Sigma(\alpha) \text{ otherwise} \end{array}$$

Fig. 8. A closer approximation of SML/NJ's application typing rule.

for type-checking *let* expressions. One generalizes both applicative and imperative type variables when the *let*-bound expression is non-expansive. The other generalizes only applicative type variables when the expression is expansive. If an expression of critical type were necessarily expansive, then $\mathrm{LET_D}$ and $\mathrm{LET_A}$ would each subsume both cases, but this is not so. For example, in the declarative systems,

$$[\,];[\,] \vdash_{[\alpha \mapsto s]} \mathit{fn}\ a \Rightarrow (\mathit{let}\ x = \mathit{ref}\ a\ \mathit{in}\ \mathit{fn}\ y \Rightarrow x)() : \alpha \rightarrow \alpha\ \mathit{ref}$$

only if $s \leq 0$. Thus the expression is of critical type but is non-expansive. Thus we conjecture that restricting any of the formalisms to using only the strengths 0 and $\infty$ (defining $0-n = 0$, $0+n = 0$, and $\infty-n = 0$, for any $n$) and augmenting it with Tofte's non-expansive *let* type inference rule is strictly more powerful than Tofte's system.

Hoang *et al.* (1993) proved the soundness of a different type system based on weak types. They permit different strengths on different instances of a type variable in a type, as in

```
fn f => f nil : ('sa list -> 'sa) -> '(s-1)a
```

for any strength $s$. The decremented strength of the function's range reflects the single application in the function body. This generalization of the SML/NJ approach gives a more informative analysis of strengths, even for purely functional terms as above, which eliminates the need for the conservative approximation of strengths at function applications. As a result, they claim that their system is more general than that of SML/NJ and provide empirical evidence of this, but they lack a formalization of SML/NJ to prove the claim.

In their analysis of reference creation, both weak and imperative type systems label type variables with information. Another approach is to label each type arrow with an *effect*, which describes an approximation of the change in the store that occurs when applying the function. The static semantics then derives both a type and an effect for an expression, and generalization is defined relative to those effects. This approach is taken by Leroy and Weis (1991), Talpin and Jouvelot (1992a; 1992b) and Wright (1992). A slightly different approach is given by Leroy (1992), where type arrows are labelled with the types of any values that may occur in references. Damas' (1985) system has aspects of both Tofte's and those using effects, as it distinguishes references that have been created from those that may be created after further application. Reynolds (1989) uses an effects-like system to detect interference such as aliasing of references.

For a comparison of some of these systems to each other, see Wright (1992), Leroy (1992) and O'Toole (1989). However, O'Toole incorrectly allows generalization of critical type variables in his formalization of weak polymorphism. Existing weak type and effect systems are formally incomparable. Effects systems generally have simpler inference rules, but in practice the approach may be unwieldy, because of the size of the type arrow labels.

Yet another alternative is to restrict polymorphic generalization to only those expressions which are values. Leroy's polymorphism by name (1992; 1993) effectively restricts generalization to thunks. Wright (1993) gives empirical evidence that imposing this restriction directly on SML is not a great sacrifice in programming flexibility, since any non-value of functional type that does not create a reference cell when evaluated may be replaced by its $\eta$-expansion, which is a value.

## 8 Conclusions and future work

We have motivated and defined several formalisms of weak polymorphic types and described their similarity to that of SML/NJ. In particular, the algorithmic family of static semantics closely model the details of the implementation. Naturally, either of those shown sound could be incorporated into SML/NJ to restore proven soundness to its type system, although it should be verified that extending the system with continuations and exceptions is still sound. The soundness of $\lambda\Psi$ and $\lambda\Psi^+$ should also be determined, since they are the most similar to the SML/NJ's implementation.

These systems provide the first formalization of weak types that are directly related to SML/NJ. As such, they should be formally compared to that of Hoang *et al.* to test their claim that their approach is strictly more general than that of SML/NJ.

From the given examples, it should be clear that these systems are complex and sometimes result in non-intuitive maximal typings. Both of these properties are undesirable, especially when types must be given in module specifications. Therefore, it is the author's opinion that such systems are not wholly suitable for practical use.

Despite the similarities to SML/NJ's implementation, detailed comparisons are still somewhat difficult because the implementation has a broader definition of occurrences and uses side effects. The $\lambda\Psi$ family of formalisms could be enriched with the more general definition of an occurrence to further study some of these details. Doing so, however, only further complicates the static semantics.

We have not explored type inference algorithms for these systems. The standard algorithm for SML (Damas and Milner, 1982) depends on the existence of principal types. We conjecture that these systems all have principal types and that $\lambda\Sigma^+$ and $\lambda\Psi^+$ have principal strengths because of the WEAKEN$_D$ rule. The other systems do not have principal strengths, but instead appear to have a maximal finite strength and possibly an infinite strength for each type variable of the principal type. Since the strengths of different type variables in a type are independent, this still gives a small number of maximal strengths.

The connection between type systems which label type variables and those which label type arrows should also be further explored. Since specific systems of these two approaches are generally incomparable in power, it may be worthwhile to somehow combine the ideas in one system. However, such a combination would likely result in types too cumbersome in practice.

## Appendix: Soundness proof for $\lambda\Sigma$

Now we present the proof of the soundness of system $\lambda\Sigma$, as outlined in section 4. Here the theorems and lemmas are in the order of their dependence.

The first lemma shows that in any derivable judgment, the strength context gives the strength of any type variable mentioned in the type assumptions or the expression's type.

**Lemma 1 (Well-formedness)**
*If* $\Lambda;\Gamma \vdash_\Sigma e : \tau$, *then* $FTV(\Lambda, \Gamma, \tau)$ *is a subset of* $dom(\Sigma)$.

**Proof** This holds by structural induction on the derivation. The side condition of each of the base cases states that $FTV(\Lambda, \Gamma)$ is a subset of $dom(\Sigma)$. And $FTV(\tau)$ is also a subset of $dom(\Sigma)$ in $\mathrm{VAR_D}$ by the definition of instantiation, in $\mathrm{LOC_D}$ since $FTV(\tau)$ is a subset of $FTV(\Lambda)$, and in $\mathrm{UNIT_D}$ since $FTV(\tau)$ is empty. The remaining cases follow inductively. □

Lemmas 2 and 3 are two common forms of type substitutions which are not simply special cases of the general Type Substitution lemma (Lemma 7) for two reasons: they do not weaken the strength context of the type derivation, and the proof of that lemma is dependent on these two. The first shows that type variables can be renamed to avoid conflicts with other strength contexts. The second shows that type variables can be replaced by arbitrary ground types to remove them from consideration.

**Lemma 2 ($\alpha$-Renaming type substitution)**
*If*
1. $\Lambda;\Gamma \vdash_\Sigma e : \tau$,
2. $S = [\alpha_1 \mapsto \beta_1, \ldots, \alpha_n \mapsto \beta_n]$, *where* $\beta_1, \ldots, \beta_n$ *are distinct from each other,*
3. $dom(S)$ *is a subset of* $dom(\Sigma)$, *and*
4. *none of* $\beta_1, \ldots, \beta_n$ *are in* $dom(\Sigma)$,

*then* $S(\Lambda); S(\Gamma) \vdash_{S(\Sigma)} e : S(\tau)$.

**Lemma 3 (Ground type substitution)**
*If*
1. $\Lambda;\Gamma \vdash_\Sigma e : \tau$,
2. $S = [\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$,
3. $dom(S)$ *is a subset of* $dom(\Sigma)$, *and*
4. $\tau_1, \ldots, \tau_n$ *are ground,*

*then* $S(\Lambda); S(\Gamma) \vdash_\Sigma e : S(\tau)$.

**Proof** Each of these lemmas holds by structural induction on its type derivation. The $\mathrm{VAR_D}$ cases are simplifications of that of Lemma 7, and the other cases follow inductively. □

The following weakening lemma shows that unnecessary type assumptions can be removed and variables added to the strength context. It also shows that finite strengths can be lowered arbitrarily. By the definition of type-matching, this lemma also extends to that relation.

**Lemma 4 (Weakening)**
*If* 1. $\Lambda;\Gamma \vdash_{\Sigma_0} e : \tau$,
2. $\Sigma'_0 \leq_\infty \Sigma_0$,
3. *$FTV(\Lambda', \Gamma')$ is a subset of $dom(\Sigma_0 \cdot \Sigma_1)$, and*
4. *$\Lambda \cdot \Lambda'$ and $\Gamma \cdot \Gamma'$ are defined,* i.e. $dom(\Lambda) \cap dom(\Lambda') = dom(\Gamma) \cap dom(\Gamma') = \emptyset$,
*then* $\Lambda \cdot \Lambda'; \Gamma \cdot \Gamma' \vdash_{\Sigma'_0 \cdot \Sigma_1} e : \tau$.

**Proof** This holds by structural induction on the type derivation. In the LET$_D$ case, we use Lemmas 2 and 3 to rename and remove type variables in $dom(\Lambda')$ that conflict with those bound in the type scheme to ensure that the strengths of the bound type variables are not lowered. □

Conversely, the next lemma allows unnecessary assumptions to be removed from the variable type assumption and the strength context. It could easily be generalized to strengthen the location type assumption as well, but that is not needed. The second and third hypotheses define when the strength context $\Sigma'$ and variable type assumption $\Gamma'$ are unnecessary.

**Lemma 5 (Strengthening)**
*If* 1. $\Lambda;\Gamma \cdot \Gamma' \vdash_{\Sigma \cdot \Sigma'} e : \tau$,
2. *$FTV(\Lambda, \Gamma, \tau)$ is a subset of $dom(\Sigma)$, and*
3. *the free variables of e are not in $dom(\Gamma')$,*
*then* $\Lambda;\Gamma \vdash_\Sigma e : \tau$.

**Proof** This holds by structural induction on the type derivation, using Lemma 3 to eliminate any extra type variables occurring only in the mediating types in $:=_D$, APP$_D$, and LET$_D$. The last hypothesis is used in the VAR$_D$ case.

For brevity, we prove only the most involved case, that of LET$_D$, which uses all of the techniques needed to prove the other cases. Inversion of the typing derivation provides $\Sigma_0$ and $\tau'$ such that $NonCrit(\Sigma_0)$, and

$$\Lambda;\Gamma \cdot \Gamma' \vdash_{\Sigma \cdot \Sigma_0 \cdot \Sigma'} e_1 : \tau' \qquad \Lambda;\Gamma \cdot \Gamma'[x : \forall \Sigma_0.\tau'] \vdash_{\Sigma \cdot \Sigma'} e_2 : \tau$$

$$Weaker(\Sigma \cdot \Sigma' \downarrow (FTV(\tau') \cap dom(\Sigma \cdot \Sigma')) \setminus FTV(\Lambda, \Gamma \cdot \Gamma'), 0). \tag{1}$$

Since $\alpha$-convertible values are identified, we can assume that the bound variable $x$ is not in $dom(\Gamma')$. Induction cannot be used yet, since $FTV(\tau')$ is not necessarily a subset of $dom(\Sigma \cdot \Sigma_0)$. So, define $S$ to be a ground type substitution on the type variables in $FTV(\tau') \cap dom(\Sigma')$. By Lemma 1 and since the domains of $\Sigma$ and $\Sigma'$ do not intersect, then $\Lambda$, $\Gamma$, and $\tau$ are invariant under $S$, and $S(\forall \Sigma_0.\tau') = \forall \Sigma_0.S(\tau')$. So, Lemma 3 gives

$$\Lambda;\Gamma \cdot S(\Gamma') \vdash_{\Sigma \cdot \Sigma_0 \cdot \Sigma'} e_1 : S(\tau') \qquad \Lambda;\Gamma \cdot S(\Gamma')[x : \forall \Sigma_0.S(\tau')] \vdash_{\Sigma \cdot \Sigma'} e_2 : \tau.$$

Induction on each of the typing derivations for $e_1$ and $e_2$ then applies, showing that

$$\Lambda;\Gamma \vdash_{\Sigma \cdot \Sigma_0} e_1 : S(\tau') \qquad \Lambda;\Gamma[x : \forall \Sigma_0.S(\tau')] \vdash_\Sigma e_2 : \tau.$$

We now want to work towards applying LET$_D$, but its side condition does not hold for these derivations as there may be a finite positive type variable in both

https://doi.org/10.1017/S0956796800001593 Published online by Cambridge University Press

$S(\tau')$ and $\Gamma'$. So, let $S'$ be a ground type substitution on the type variables in $(FTV(S(\tau')) \cap FTV(\Gamma') \cap dom(\Sigma)) \setminus FTV(\Lambda, \tau)$. By the construction of $S'$ and the second hypothesis, then $\Lambda$, $\Gamma$, and $\tau$ are all invariant under $S'$, and $dom(S')$ is a subset of $dom(\Sigma, \Sigma_0)$. So, Lemma 3 on each of the previous typing derivations gives

$$\Lambda; \Gamma \vdash_{\Sigma \cdot \Sigma_0} e_1 : S'(S(\tau')) \qquad \Lambda; \Gamma[x : S'(\forall \Sigma_0 . S(\tau'))] \vdash_\Sigma e_2 : \tau.$$

And note that since $dom(S')$ is a subset of $dom(\Sigma)$, it shares no variables with $\Sigma_0$, so $S'(\forall \Sigma_0 . S(\tau')) = \forall \Sigma_0 . S'(S(\tau'))$. The appropriate side condition of $\mathrm{LET_D}$,

$$Weaker(\Sigma \downarrow (FTV(S'(S(\tau'))) \cap dom(\Sigma)) \setminus FTV(S'(S(\Lambda)), S'(S(\Gamma))), 0),$$

now holds by Statement 1 and since for any ground type substitution $S$ and sets $A$ and $B$, $FTV(S(A)) \setminus FTV(S(B))$ is a subset of $FTV(A) \setminus FTV(B)$. Therefore the conclusion holds by $\mathrm{LET_D}$. □

The following lemma allows the strength context to be strengthened by removing irrelevant type variables. The converse, adding irrelevant type variables, is a special case of Lemma 4. Together they show that the strength of type variables not in a typing judgment's type assumptions or type may be changed freely.

**Lemma 6 (Strength context strengthening)**
*If* $\Lambda; \Gamma \vdash_{\Sigma \cdot \Sigma'} e : \tau$, *and* $FTV(\Lambda, \Gamma, \tau)$ *is a subset of* $dom(\Sigma)$, *then* $\Lambda; \Gamma \vdash_\Sigma e : \tau$.

**Proof** The proof is by structural induction on the typing derivation. As in previous lemmas, the $:=_\mathrm{D}$, $\mathrm{APP_D}$, and $\mathrm{LET_D}$ cases use Lemma 3 to eliminate any type variables which occur only in intermediate stages of the derivation. □

**Lemma 7 (Type substitution)**
*If*
1. $\Lambda; \Gamma \vdash_{(\Sigma \cdot \Sigma_1 \cdot \Sigma_0) + c'} e : \tau$,
2. $S = [\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$,
3. $dom(S) = dom(\Sigma_1)$,
4. *for all* $i$ *in* $\{1, \ldots, n\}$, $Crit((\Sigma \cdot \Sigma_2) + c - \Sigma_1(\alpha_i) \downarrow FTV(\tau_i))$, *and*
5. $Weaker(\Sigma \downarrow FTV(\Lambda, \Gamma, \tau) \cap dom(\Sigma), 0)$,

*then there exists* $\Sigma'$ *such that* $\Sigma' \leq_1 \Sigma$, *and* $S(\Lambda); S(\Gamma) \vdash_{(((\Sigma' \cdot \Sigma_2) + c) \cdot \Sigma_0) + c'} e : S(\tau)$.

**Proof** This holds by structural induction on the typing derivation. The constant $c$ and strength context $\Sigma_1$ are those used in Lemma 8. The constant $c'$ allows the first hypothesis to hold inductively in the $\mathrm{APP_D}$ case, and similarly $\Sigma_0$ is for the $\mathrm{LAM_D}$ and $\mathrm{LET_D}$ cases. We will use only the case where $c' = 0$, and $\Sigma_0 = []$.

The $\mathrm{LOC_D}$ and $\mathrm{UNIT_D}$ cases hold trivially with the definition $\Sigma' = \Sigma$. For the remaining cases, note that $FTV(\tau_1, \ldots, \tau_n)$ is a subset of $dom(\Sigma \cdot \Sigma_2)$ by the fourth hypothesis.

In the $\mathrm{VAR_D}$ case, inversion of the type derivation states that the instantiation $\vdash_{(\Sigma \cdot \Sigma_1 \cdot \Sigma_0) + c'} \Gamma(x) \geq \tau$ holds. We work towards showing the instantiation necessary for the conclusion. Let

$$\Gamma(x) = \forall [\beta_1 \mapsto s_1, \ldots, \beta_k \mapsto s_k] . \tau'.$$

By $\alpha$-equivalence of type schemes, we can assume that the domain of $S$ does not

https://doi.org/10.1017/S0956796800001593 Published online by Cambridge University Press

overlap with any of $\beta_1, \ldots, \beta_k$, so

$$S(\Gamma(x)) = \forall[\beta_1 \mapsto s_1, \ldots, \beta_k \mapsto s_k].S(\tau').$$

By the definition of instantiation, there exists a type substitution

$$S' = [\beta_1 \mapsto \tau'_1, \ldots, \beta_k \mapsto \tau'_k]$$

such that $S'(\tau') = \tau$. In particular, we can choose $S'$ such that for all $j$ in $\{1, \ldots, k\}$, if $\beta_j$ is not in $FTV(\tau')$, then $\tau'_j$ is ground, so that $FTV(\tau'_1, \ldots, \tau'_k)$ is a subset of $FTV(\tau)$. Now define $\Sigma'$ so that for all $\alpha$ in $dom(\Sigma') = dom(\Sigma)$,

$$\Sigma'(\alpha) = \begin{cases} \min(\Sigma(\alpha), -c) & \text{if } \alpha \text{ is in } FTV(\tau'_1, \ldots, \tau'_k), \\ \Sigma(\alpha) & \text{otherwise.} \end{cases}$$

By the last hypothesis, and since $FTV(\tau'_1, \ldots, \tau'_k)$ is a subset of $FTV(\tau)$, then $\Sigma' \leq_1 \Sigma$. And since the definition of instantiation implies that for all $j$ in $\{1, \ldots, k\}$,

$$Crit\,((\Sigma \cdot \Sigma_1 \cdot \Sigma_0) + c' - s_j \downarrow FTV(\tau'_j)),$$

then the fourth hypothesis implies that for all $j$ in $\{1, \ldots, k\}$,

$$Crit\,((((\Sigma' \cdot \Sigma_2) + c) \cdot \Sigma_0) + c' - s_j \downarrow FTV(S(\tau'_j))).$$

Thus the instantiation $\vdash_{(((\Sigma' \cdot \Sigma_2)+c) \cdot \Sigma_0)+c'} S(\Gamma(x)) \succeq S(\tau)$ holds, and the conclusion follows by $\mathrm{VAR_D}$.

The $!_\mathrm{D}$ and $\mathrm{LAM_D}$ cases follow simply by induction. Both the $:=_\mathrm{D}$ and $\mathrm{APP_D}$ cases must also use Lemma 3, while in the $\mathrm{REF_D}$, $\mathrm{APP_D}$, and $\mathrm{LET_D}$ cases, an appropriate $\Sigma'$ must be calculated to satisfy the side conditions, as in the $\mathrm{VAR_D}$ case.

For example, in the $\mathrm{APP_D}$ case, inversion gives a $\tau'$ such that

$$\Lambda; \Gamma \vdash_{(\Sigma \cdot \Sigma_1 \cdot \Sigma_0)+c'+1} e_1 : \tau' \rightarrow \tau \qquad \Lambda; \Gamma \vdash_{(\Sigma \cdot \Sigma_1 \cdot \Sigma_0)+c'} e_2 : \tau'$$

$$Weaker\,((\Sigma \cdot \Sigma_1 \cdot \Sigma_0) + c' \downarrow FTV(\tau'), 0). \tag{2}$$

To allow induction to be used on these type derivations, we must remove the type variables which occur only in the mediating type $\tau'$, so that the last hypothesis holds inductively. So, define a ground type substitution $S'$ over the type variables in $(FTV(\tau') \cap dom(\Sigma)) \setminus FTV(\Lambda, \Gamma, \tau)$. In particular, $\Lambda$, $\Gamma$, and $\tau$ are unaffected by $S'$, so by Lemma 3,

$$\Lambda; \Gamma \vdash_{(\Sigma \cdot \Sigma_1 \cdot \Sigma_0)+c'+1} e_1 : S'(\tau') \rightarrow \tau \qquad \Lambda; \Gamma \vdash_{(\Sigma \cdot \Sigma_1 \cdot \Sigma_0)+c'} e_2 : S'(\tau').$$

By the construction of $S'$, $FTV(S'(\tau')) \cap dom(\Sigma)$ is a subset of $FTV(\Lambda, \Gamma, \tau)$. So, with the last hypothesis, we have

$$Weaker\,(\Sigma \downarrow FTV(\Lambda, \Gamma, \tau, S'(\tau')) \cap dom(\Sigma), 0).$$

Thus induction can be applied on the type derivations of $e_1$ and $e_2$, proving that there exist $\Sigma''$ and $\Sigma'''$ such that $\Sigma'' \leq_1 \Sigma$, $\Sigma''' \leq_1 \Sigma$, and

$$S(\Lambda); S(\Gamma) \vdash_{(((\Sigma'' \cdot \Sigma_2)+c) \cdot \Sigma_0)+c'+1} e_1 : S(S'(\tau') \rightarrow \tau)$$

$$S(\Lambda); S(\Gamma) \vdash_{(((\Sigma''' \cdot \Sigma_2)+c) \cdot \Sigma_0)+c'} e_2 : S(S'(\tau')).$$

Now we work towards applying the $\mathrm{APP_D}$ rule. To use the same strength context in these derivations, let $\Sigma'''' = \min(\Sigma'', \Sigma''')$, where the minimum of two strength contexts having the same domain is defined by the point-wise minimum. Clearly, $\Sigma'''' \leq_1 \Sigma$. To satisfy $\mathrm{APP_D}$'s side condition, define $\Sigma'$ so that for all $\alpha$ in $dom(\Sigma') = dom(\Sigma)$,

$$\Sigma'(\alpha) = \begin{cases} \min(\Sigma''''(\alpha), -c) & \text{if } \alpha \text{ is in } FTV(S'(\tau')), \\ \Sigma''''(\alpha) & \text{otherwise.} \end{cases}$$

Since $Weaker(\Sigma \downarrow FTV(S'(\tau')) \cap dom(\Sigma), 0)$, the last hypothesis and Statement 2 show that $\Sigma' \leq_1 \Sigma''''$. Thus the above typing derivations for $e_1$ and $e_2$ can be weakened to use $\Sigma'$ in place of $\Sigma''$ and $\Sigma'''$. And since $FTV(S'(\tau'))$ is a subset of $FTV(\tau')$, the necessary side condition holds:

$$Weaker((((\Sigma' \cdot \Sigma_2) + c) \cdot \Sigma_0) + c' \downarrow FTV(S(S'(\tau'))), 0).$$

For the type variables of $S'(\tau')$ that are in $\Sigma'$, this follows from the construction of $\Sigma'$ and Statement 2. Those in $\Sigma_1$ were substituted by $S$, and this holds by the fourth hypothesis. And for those in $\Sigma_0$, this follows directly from Statement 2. Thus the conclusion holds by $\mathrm{APP_D}$. □

**Lemma 8 (Value substitution)**
*If* 1. $\Lambda; \Gamma[x : \forall\Sigma_2.\tau_2] \vdash_{(\Sigma\cdot\Sigma_1)+c} e : \tau_1$,
2. $\Lambda; [\,] \vdash_{\Sigma\cdot\Sigma_2} v : \tau_2$,
3. $Weaker(\Sigma \downarrow FTV(\Lambda, \tau_2) \cap dom(\Sigma), 0)$,

*then there exists a* $\Sigma'$ *such that* $\Sigma' \leq_1 \Sigma$, *and* $\Lambda; \Gamma \vdash_{(\Sigma'\cdot\Sigma_1)+c} [v/x]e : \tau_1$.

**Proof** The proof is by structural induction on the type derivation for $e$. The constant $c$ is used to allow the first hypothesis to hold inductively in the $\mathrm{APP_D}$ case, and similarly $\Sigma_1$ is used for the $\mathrm{LET_D}$ case, and $\Gamma$ for $\mathrm{APP_D}$ and $\mathrm{LET_D}$. We are only interested in the case where $c = 0$, $\Sigma_1 = [\,]$, and $\Gamma = [\,]$.

The $\mathrm{LOC_D}$, $\mathrm{UNIT_D}$, and when $e \neq x$, $\mathrm{VAR_D}$ cases follow from Lemma 5 with the definition $\Sigma' = \Sigma$. When $e = x$, then $\vdash_{(\Sigma\cdot\Sigma_1)+c} \forall\Sigma_2.\tau_2 \succeq \tau_1$ follows by inversion. By this instantiation, there is a type substitution $S$ such that $S(\tau_2) = \tau_1$, and for all $\alpha$ in $dom(S) = dom(\Sigma_2)$, $Crit((\Sigma \cdot \Sigma_1) + c - \Sigma_2(\alpha) \downarrow FTV(S(\alpha)))$. By Lemma 1 applied to the first hypothesis, $FTV(\Lambda)$ is a subset of $dom(\Sigma \cdot \Sigma_1)$, and thus $\Lambda$ is unaffected by $S$. The conclusion then holds by Lemmas 7 and 5.

The $\mathrm{REF_D}$, $!_\mathrm{D}$, and $\mathrm{LAM_D}$ cases hold by induction, while the $:=_\mathrm{D}$ and $\mathrm{APP_D}$ cases also use Lemma 4. In the $\mathrm{LET_D}$ case, we must calculate an appropriate $\Sigma'$. Inversion of the first hypothesis shows there are $\tau'$ and $\Sigma_0$ such that $NonCrit(\Sigma_0)$, and

$$\Lambda; \Gamma[x : \forall\Sigma_2.\tau_2] \vdash_{((\Sigma\cdot\Sigma_1)+c)\cdot\Sigma_0} e_1 : \tau'$$

$$\Lambda; \Gamma[x : \forall\Sigma_2.\tau_2, y : \forall\Sigma_0.\tau'] \vdash_{(\Sigma\cdot\Sigma_1)+c} e_2 : \tau_1$$

$$Weaker((\Sigma' \cdot \Sigma_1) + c \downarrow (FTV(\tau') \cap dom(\Sigma \cdot \Sigma_1)) \setminus FTV(\Lambda, \Gamma[x : \forall\Sigma_2.\tau_2]), 0).$$

By Lemma 2, we can assume that the type variables in $\Sigma_0$ do not clash with those in $\Sigma_1$. So, by induction on the type derivations of $e_1$ and $e_2$, there exist $\Sigma''$ and $\Sigma'''$ such that $\Sigma'' \leq_1 \Sigma$, $\Sigma''' \leq_1 \Sigma$, and

$$\Lambda; \Gamma \vdash_{((\Sigma''\cdot\Sigma_1)+c)\cdot\Sigma_0} [v/x]e_1 : \tau' \qquad \Lambda; \Gamma[y : \forall\Sigma_0.\tau'] \vdash_{(\Sigma'''\cdot\Sigma_1)+c} [v/x]e_2 : \tau_1.$$

Let $\Sigma'''' = \min(\Sigma'', \Sigma''')$. In order to satisfy the last side condition of $\mathrm{LET_D}$, define $\Sigma'$ so that for all $\alpha$ in $dom(\Sigma') = dom(\Sigma)$,

$$\Sigma'(\alpha) = \begin{cases} \min(\Sigma''''(\alpha), -c) & \text{if } \alpha \text{ is in } FTV(\tau_1), \\ \Sigma''''(\alpha) & \text{otherwise.} \end{cases}$$

By the last hypothesis, $\Sigma' \leq_1 \Sigma$, and $Weaker(\Sigma' + c \downarrow FTV(\forall\Sigma_2.\tau_2), 0)$. So the side condition

$$Weaker((\Sigma \cdot \Sigma_1) + c \downarrow (FTV(\tau') \cap dom(\Sigma \cdot \Sigma_1)) \setminus FTV(\Lambda, \Gamma), 0)$$

holds, and the conclusion follows by weakening the type derivations to use $\Sigma'$, and then using $\mathrm{LET_D}$. □

**Theorem 1 (Type preservation under evaluation)**
*If* 1. $\sigma \vdash e \Longrightarrow v, \sigma'$,
2. $\Lambda; [\,] \vdash_{(\Sigma+c)\cdot\Sigma_1} e : \tau$,
3. $\vdash_\Sigma \sigma : \Lambda$,
4. $Crit(\Sigma \downarrow FTV(\Lambda))$,
5. $NonCrit(\Sigma_1)$, *and*
6. $c \geq 0$,

*then there exists* $\Lambda_0$ *and* $\Sigma_0$ *such that*

1. $\Sigma_0 \leq_1 \Sigma$
2. $\Lambda \cdot \Lambda_0; [\,] \vdash_{(\Sigma_0+c)\cdot\Sigma_1} v : \tau$,
3. $\vdash_{\Sigma_0} \sigma' : \Lambda \cdot \Lambda_0$, *and*
4. $Crit(\Sigma_0 + c \downarrow FTV(\Lambda_0))$

**Proof** The proof is by structural induction on the evaluation derivation. The constant $c$ generalizes the constant 1 added in $\mathrm{APP_D}$ and corresponds with the abstraction depth of the algorithmic formalisms. Thus the strength context $\Sigma$ is that of the top-level. The strength context $\Sigma_1$ generalizes the local strength context used in $\mathrm{LET_D}$ and is kept separate from $\Sigma$ so that it does not interact with the constant $c$. The Top-Level Type Preservation Under Evaluation theorem then holds by setting $c = 0$, $\Lambda = [\,]$, $\sigma = [\,]$, and $\Sigma = [\,]$.

The VAL case holds with $\Lambda_0 = [\,]$, and $\Sigma_0 = \Sigma$, since $\sigma' = \sigma$.

Since the location type assumption and strength context are extended and weakened for each use of induction, Lemma 4 is used frequently in the inductive cases. For the ALLOC case, inversion of the first two hypotheses gives

$$\sigma \vdash e \Longrightarrow v, \sigma_1 \qquad \sigma' = \sigma_1[l \mapsto v] \qquad \Lambda; [\,] \vdash_{(\Sigma+c)\cdot\Sigma_1} e : \tau'$$

$$Crit((\Sigma + c) \cdot \Sigma_1 \downarrow FTV(\tau')),$$

where $\tau = \tau'$ *ref*. So, induction on the evaluation derivation of $e$ shows there exist $\Lambda_1$ and $\Sigma_0$ such that the first conclusion holds, and

$$\Lambda \cdot \Lambda_1; [\,] \vdash_{(\Sigma_0+c)\cdot\Sigma_1} v : \tau' \qquad \vdash_{\Sigma_0} \sigma_1 : \Lambda \cdot \Lambda_1 \qquad Crit(\Sigma_0 + c \downarrow FTV(\Lambda_1)).$$

Defining $\Lambda_0 = \Lambda_1[l : \tau]$, then $\Lambda \cdot \Lambda_0; [\,] \vdash_{(\Sigma_0+c)\cdot\Sigma_1} l : \tau'$ *ref* holds by LOC. So the third conclusion holds by Lemma 4 and the definition of type-matching. The remaining conclusions hold since $FTV(\Lambda_0) = FTV(\Lambda_1, \tau')$, and $Crit(\Sigma + c \downarrow FTV(\tau'))$.

https://doi.org/10.1017/S0956796800001593 Published online by Cambridge University Press

In the CONT case, inversion gives

$$\Lambda; [] \vdash_{(\Sigma+c)\cdot\Sigma_1} e : \tau\ ref \qquad \sigma \vdash e \Longrightarrow l, \sigma',$$

where $v = \sigma'(l)$. Induction on the evaluation derivation of $e$ then proves there exists a $\Lambda_0$ and $\Sigma_1$ such that $\Sigma' \leq_1 \Sigma$, and

$$\Lambda \cdot \Lambda_0; [] \vdash_{(\Sigma'+c)\cdot\Sigma_1} l : \tau\ ref \qquad \vdash_{\Sigma'} \sigma' : \Lambda \cdot \Lambda_0 \qquad Crit(\Sigma' + c \downarrow FTV(\Lambda_0)).$$

So, $\Lambda \cdot \Lambda_0(l) = \tau$. By the definition of type-matching and Lemma 5, then we have $\Lambda \cdot \Lambda_0; [] \vdash_{\Sigma'} v : \tau$. Define $\Sigma_0$ so that for all $\alpha$ in $dom(\Sigma_0) = dom(\Sigma)$,

$$\Sigma_0(\alpha) = \begin{cases} \Sigma'(\alpha) - c & \text{if } \alpha \text{ is in } FTV(\Lambda \cdot \Lambda_0), \\ \Sigma'(\alpha) & \text{otherwise.} \end{cases}$$

Since $Crit(\Sigma' \downarrow FTV(\Lambda \cdot \Lambda_0))$, then the first and fourth conclusions hold. The second follows by Lemmas 6 and 4 since $\Sigma_0 + c$ and $\Sigma'$ differ only for irrelevant type variables. And the third follows by Lemma 4.

The UPD, APPLY, and BIND cases are similar, with a pattern of induction followed by weakening. The latter two also use Lemma 8 to allow induction on the result of substitution.

For APPLY, inversion of these first two hypotheses gives $\sigma_1$ and $\sigma_2$ such that

$$\sigma \vdash e_1 \Longrightarrow fn\ x \Rightarrow e_1', \sigma_1 \qquad \sigma_1 \vdash e_2 \Longrightarrow v_2, \sigma_2 \qquad \sigma_2 \vdash [v_2/x]e_1' \Longrightarrow v, \sigma',$$

and $\tau'$ such that

$$\Lambda; [] \vdash_{(\Sigma+c+1)\cdot(\Sigma_1+1)} e_1 : \tau' \to \tau \qquad \Lambda; [] \vdash_{(\Sigma+c)\cdot\Sigma_1} e_2 : \tau'$$

$$Weaker((\Sigma + c) \cdot \Sigma_1 \downarrow FTV(\tau'), 0).$$

Induction on the $e_1$ evaluation derivation shows that there exist $\Lambda_1$ and $\Sigma'$ such that $\Sigma' \leq_1 \Sigma$, and

$$\Lambda \cdot \Lambda_1; [] \vdash_{(\Sigma'+c+1)\cdot(\Sigma_1+1)} fn\ x \Rightarrow e_1' : \tau' \to \tau \qquad \vdash_{\Sigma'} \sigma_1 : \Lambda \cdot \Lambda_1$$

$$Crit(\Sigma' + c + 1 \downarrow FTV(\Lambda_1)).$$

To apply induction to the $e_2$ evaluation deriviation, we first note that we have $Crit(\Sigma' + c \downarrow FTV(\Lambda \cdot \Lambda_1))$, and that Lemma 4 proves $\Lambda \cdot \Lambda_1; [] \vdash_{(\Sigma'+c)\cdot\Sigma_1} e_2 : \tau'$. Induction then gives $\Lambda_2$ and $\Sigma'$ such that $\Sigma'' \leq_1 \Sigma'$, and

$$\Lambda \cdot \Lambda_1 \cdot \Lambda_2; [] \vdash_{(\Sigma''+c)\cdot\Sigma_1} v_2 : \tau' \qquad \vdash_{\Sigma''} \sigma_2 : \Lambda \cdot \Lambda_1 \cdot \Lambda_2$$

$$Crit(\Sigma'' + c \downarrow FTV(\Lambda_2)).$$

We cannot yet apply Lemma 8 to obtain a typing of the appropriate substitution. To satisfy that lemma's third hypothesis, define $\Sigma_3$ so that for all $\alpha$ in $dom(\Sigma''') = dom(\Sigma'')$,

$$\Sigma'''(\alpha) = \begin{cases} \min(\Sigma''(\alpha), -c) & \text{if } \alpha \text{ is in } FTV(\Lambda), \\ \Sigma''(\alpha) & \text{otherwise.} \end{cases}$$

Thus $\Sigma''' \leq_1 \Sigma''$, and $Weaker((\Sigma''' + c) \cdot \Sigma_1 \downarrow FTV(\Lambda \cdot \Lambda_1 \cdot \Lambda_2, \tau'), 0)$. Inversion on the type derivation on the function value and Lemma 4 prove

$$\Lambda \cdot \Lambda_1 \cdot \Lambda_2; [x : \tau'] \vdash_{(\Sigma'''+c)\Sigma_1} e_1' : \tau \qquad \Lambda \cdot \Lambda_1 \cdot \Lambda_2; [] \vdash_{(\Sigma'''+c)\cdot\Sigma_1} v_2 : \tau'.$$

So, using the trivial generalization of $\tau'$, Lemma 8 proves that there exists a $\Sigma''''$ such that $\Sigma'''' + c \leq_1 \Sigma''' + c$, and

$$\Lambda \cdot \Lambda_1 \cdot \Lambda_2 ; [] \vdash_{(\Sigma''''+c)\cdot\Sigma_1} [v_2/x]e_1' : \tau.$$

We do not need to weaken $\Sigma_1$ here since it is non-critical. Since $c \geq 0$, then $\Sigma'''' \leq_1 \Sigma'''$, and $Crit(\Sigma'''' \downarrow FTV(\Lambda \cdot \Lambda_1 \cdot \Lambda_2))$. We can now apply induction to the final evaluation derivation, obtaining $\Lambda_3$ and $\Sigma_0$ such that $\Sigma_0 \leq_1 \Sigma''''$, and

$$\Lambda \cdot \Lambda_1 \cdot \Lambda_2 \cdot \Lambda_3 ; [] \vdash_{(\Sigma_0+c)\cdot\Sigma_1} v : \tau \qquad \vdash_{\Sigma_0} \sigma' : \Lambda \cdot \Lambda_1 \cdot \Lambda_2 \cdot \Lambda_3$$

$$Crit(\Sigma_0 + c \downarrow FTV(\Lambda_3)).$$

The conclusions then hold with $\Lambda_0 = \Lambda_1 \cdot \Lambda_2 \cdot \Lambda_3$.

The BIND case is very similar, but with only two uses of induction. Inversion of the first two hypotheses proves there exist $\sigma_1$ and $v_1$ such that

$$\sigma \vdash e_1 \Longrightarrow v_1, \sigma_1 \qquad \sigma_1 \vdash [v_1/x]e_2 \Longrightarrow v_2, \sigma_2,$$

and $\Sigma_2$ and $\tau'$ such that $NonCrit(\Sigma_2)$, and

$$\Lambda ; [] \vdash_{(\Sigma+c)\cdot\Sigma_1\cdot\Sigma_2} e_1 : \tau' \qquad \Lambda ; [x : \forall\Sigma_2.\tau'] \vdash_{(\Sigma+c)\cdot\Sigma_1} e_2 : \tau$$

$$Weaker((\Sigma + c) \cdot \Sigma_1 \downarrow (FTV(\tau') \cap dom(\Sigma \cdot \Sigma_1)) \setminus FTV(\Lambda, []), 0).$$

Then induction on the $e_1$ evaluation derivation shows there exist $\Lambda_1$ and $\Sigma'$ such that $\Sigma' \leq_1 \Sigma$, and

$$\Lambda \cdot \Lambda_1 ; [] \vdash_{(\Sigma'+c)\cdot\Sigma_1\cdot\Sigma_2} v_1 : \tau' \qquad \vdash_{\Sigma'} \sigma_1 : \Lambda \cdot \Lambda_1 \qquad Crit(\Sigma' + c \downarrow FTV(\Lambda_1)).$$

We cannot yet apply Lemma 8 to obtain a typing of the appropriate substitution. To satisfy that lemma's third hypothesis, define $\Sigma''$ so that for all $\alpha$ in $dom(\Sigma'') = dom(\Sigma')$,

$$\Sigma''(\alpha) = \begin{cases} \min(\Sigma'(\alpha), -c) & \text{if } \alpha \text{ is in } FTV(\Lambda), \\ \Sigma'(\alpha) & \text{otherwise,} \end{cases}$$

so that $\Sigma'' \leq_1 \Sigma'$, and $Weaker((\Sigma'' + c) \cdot \Sigma_1 \downarrow FTV(\Lambda \cdot \Lambda_1, \tau') \cap dom(\Sigma'' \cdot \Sigma_1), 0)$. And now Lemma 4 shows that $\Lambda \cdot \Lambda_1 ; [x : \forall\Sigma_2.\tau'] \vdash_{(\Sigma''+c)\cdot\Sigma_1} e_2 : \tau$, so that Lemma 8 applies, giving a $\Sigma'''$ such that $\Sigma''' \leq_1 \Sigma''$, and

$$\Lambda \cdot \Lambda_1 ; [] \vdash_{(\Sigma'''+c)\cdot\Sigma_1} [v_1/x]e_2 : \tau.$$

As in the APPLY case, we do not need to weaken the non-critical $\Sigma_1$. Next, Lemma 4 shows that $\vdash_{\Sigma''} \sigma_1 : \Lambda \cdot \Lambda_1$, and since also $Crit(\Sigma'' + c \downarrow FTV(\Lambda_1))$, then induction applies on the evaluation derivation of the substitution. This shows there are $\Lambda_2$ and $\Sigma_0$ such that $\Sigma_0 \leq_1 \Sigma'''$, and

$$\Lambda \cdot \Lambda_1 \cdot \Lambda_2 ; [] \vdash_{(\Sigma_0+c)\cdot\Sigma_1} v_2 : \tau \qquad \vdash_{\Sigma_0} \sigma_2 : \Lambda \cdot \Lambda_1 \cdot \Lambda_2 \qquad Crit(\Sigma_0 + c \downarrow FTV(\Lambda_2)).$$

The conclusion then holds by defining $\Lambda_0 = \Lambda_1 \cdot \Lambda_2$. ☐

**Theorem 2 (Well-typed programs do not go wrong)**
*If* 1. $\sigma \vdash e \Longrightarrow wrong, []$,
2. $\vdash_\Sigma \sigma : \Lambda$,
3. $Crit(\Sigma \downarrow FTV(\Lambda))$,

https://doi.org/10.1017/S0956796800001593 Published online by Cambridge University Press

*then there do* not *exist* $\Sigma_1$, *c, and* $\tau$ *such that*

1. $\Lambda; [\,] \vdash_{(\Sigma+c)\cdot\Sigma_1} e : \tau$,
2. *NonCrit* $(\Sigma_1)$, *and*
3. $c \geq 0$.

**Proof** This is proved by structural induction on the evaluation derivation. For brevity, we say in this proof simply that an expression is typable only if the three conclusions do hold. As in Theorem 1, the constant $c$ and strength context $\Sigma_1$ allow induction in the APPLY and BIND cases.

The expression $e$ cannot be a value, since values do not go wrong. So, consider the expressions that must be evaluated to evaluate $e$. For $e$ to go wrong, one of these expressions, say $e'$, goes wrong or its value is of the wrong form, e.g. a function is being dereferenced. If the former, induction shows that $e'$ must not be typable, which leads to the untypability of $e$. If the latter, $e'$ may be typable, but not so that $e$ is, e.g. the expression ! $e'$ cannot be typed when $e'$ has a function type. The following describes the cases in more detail.

If $e = \mathit{ref}\ e'$, then the only way for $e$ to go wrong is for $e'$ to go wrong. By induction, $e'$ is not typable, so neither is $e$.

If $e = !\ e'$, there are two possible ways for $e$ to go wrong. First, if $e'$ goes wrong, then by induction $e'$ is not typable, so neither is $e'$. Second, $e$ goes wrong if $\sigma \vdash e' \Longrightarrow a, \sigma'$, where $a$ is one of () or $\mathit{fn}\ x \Rightarrow e$. Assuming that $e$ is typable, i.e. $\Lambda; [\,] \vdash_{(\Sigma+c)\cdot\Sigma_1} e : \tau$, leads to a contradiction. For that typing to hold, then $\Lambda; [\,] \vdash_{(\Sigma+c)\cdot\Sigma_1} e' : \tau\ \mathit{ref}$ must hold. But by Theorem 1, then there would exist $\Sigma'$ and $\Lambda'$ such that $\Lambda \cdot \Lambda'; [\,] \vdash_{\Sigma'} a : \tau\ \mathit{ref}$, which is impossible with the two possible values of $a$. So, $e$ is not typable.

If $e = e_1{:=}e_2$, there are four possible ways for $e$ to go wrong. Three of these are like those of the previous case: $e_1$ could go wrong, $e_1$ could evaluate to () or a function, or $e_1$ could evaluate to a location not in the resulting store. The fourth case, that of $e_2$ going wrong, combines aspects of the other subcases, and we describe it further. To apply induction on the evaluation derivation of $e_2$, we must extend the type-matching to include any locations created by the evaluation of $e_1$. Assuming that $e$ is typable implies that there exists a $\tau'$ such that

$$\Lambda; [\,] \vdash_{(\Sigma+c)\cdot\Sigma_1} e_1 : \tau'\ \mathit{ref} \qquad \Lambda; [\,] \vdash_{(\Sigma+c)\cdot\Sigma_1} e_2 : \tau'.$$

By Theorem 1 on the typing of $e_1$, there exist $\Sigma'$ and $\Lambda'$ such that $\Sigma' \leq_1 \Sigma$, and

$$\vdash_{\Sigma'} \sigma_1 : \Lambda \cdot \Lambda' \qquad \mathit{Crit}\,(\Sigma' + c \downarrow \mathit{FTV}(\Lambda')),$$

and thus $\mathit{Crit}\,(\Sigma' \downarrow \mathit{FTV}(\Lambda \cdot \Lambda'))$. By Lemma 4 we have $\Lambda \cdot \Lambda'; [\,] \vdash_{(\Sigma'+c)\cdot\Sigma_1} e_2 : \tau'$, which contradicts the conclusion of applying induction on $e_2$, so the assumption that $e$ is typable must be false.

If $e = e_1\ e_2$, again we have four subcases, three of which are like those previously described: $e_1$ could go wrong, $e_2$ could go wrong, or $e_1$ could evaluate to a value of the wrong form – here, either () or a location. The fourth subcase, that of the substitution instance going wrong, is similar to those shown previously, but Theorem 1 is used on both $e_1$ and $e_2$ to obtain the required type-matching, and

Lemma 8 is used to type the substitution instance. Assuming that $e$ is typable implies that there exists a $\tau'$ such that

$$\Lambda; [\,] \vdash_{(\Sigma+c+1)\cdot(\Sigma_1+1)} e_1 : \tau' \rightarrow \tau \qquad \Lambda; [\,] \vdash_{(\Sigma+c)\cdot\Sigma_1} e_2 : \tau'.$$

Since $c + 1 \geq 0$, Theorem 1 shows there would exist $\Sigma'$ and $\Lambda'$ such that $\Sigma' \leq_1 \Sigma$, and

$$\Lambda \cdot \Lambda'; [\,] \vdash_{(\Sigma'+c+1)\cdot(\Sigma_1+1)} fn\ x \Rightarrow e_1' : \tau' \rightarrow \tau \qquad \vdash_{\Sigma'} \sigma_1 : \Lambda \cdot \Lambda'$$

$$Crit\,(\Sigma' + c + 1 \downarrow FTV(\Lambda')).$$

Then Lemma 4 would show that

$$\Lambda \cdot \Lambda'; [\,] \vdash_{(\Sigma'+c)\cdot\Sigma_1} e_2 : \tau',$$

and since $c \geq 0$, then $Crit\,(\Sigma' \downarrow FTV(\Lambda \cdot \Lambda'))$. So, Theorem 1 shows there would exist $\Sigma''$ and $\Lambda''$ such that $\Sigma'' \leq_1 \Sigma'$, and

$$\Lambda \cdot \Lambda' \cdot \Lambda''; [\,] \vdash_{(\Sigma''+c)\cdot\Sigma_1} a_2 : \tau' \qquad \vdash_{\Sigma''} \sigma_2 : \Lambda \cdot \Lambda' \cdot \Lambda''$$

$$Crit\,(\Sigma'' + c \downarrow FTV(\Lambda'')).$$

We cannot yet apply Lemma 8 to obtain a typing of the appropriate substitution. To satisfy that lemma's third hypothesis, define $\Sigma'''$ so that for all $\alpha$ in $dom\,(\Sigma''') = dom\,(\Sigma'')$,

$$\Sigma'''(\alpha) = \begin{cases} \min(\Sigma''(\alpha), -c) & \text{if } \alpha \text{ is in } FTV(\Lambda), \\ \Sigma''(\alpha) & \text{otherwise.} \end{cases}$$

Then $\Sigma''' \leq_1 \Sigma''$, and $Weaker\,((\Sigma''' + c) \cdot \Sigma_1 \downarrow FTV(\Lambda, \tau') \cap dom\,(\Sigma''' \cdot \Sigma_1), 0)$. After inversion and Lemma 4 show that $\Lambda \cdot \Lambda' \cdot \Lambda''; [x : \tau'] \vdash_{(\Sigma''+c)\cdot\Sigma_1} e_1' : \tau$, Lemma 8 shows there would exist a $\Sigma''''$ such that $\Sigma'''' + c \leq_1 \Sigma''' + c$ (and thus $\Sigma'''' \leq_1 \Sigma'''$), and

$$\Lambda \cdot \Lambda' \cdot \Lambda''; [\,] \vdash_{(\Sigma''''+c)\cdot\Sigma_1} [a_2/x]e_1' : \tau. \tag{3}$$

By Lemma 4, the type-matching above could use the same strength context, i.e. $\vdash_{\Sigma''''} \sigma_2 : \Lambda \cdot \Lambda' \cdot \Lambda''$. But since $Crit\,(\Sigma'''' \downarrow FTV(\Lambda \cdot \Lambda' \cdot \Lambda''))$, induction on the substitution instance contradicts Statement 3, so the assumption that $e$ has the typing is false.

Finally, if $e = let\ x = e_1\ in\ e_2$, then $e$ can go wrong only if $e_1$ or the appropriate substitution instance of $e_2$ goes wrong. The first possibility is like those previously shown. The second is similar to the last subcase of the previous application case. Assuming that $e$ is typable implies that there exist $\Sigma_2$ and $\tau'$ such that $NonCrit\,(\Sigma_2)$, and

$$\Lambda; [\,] \vdash_{(\Sigma+c)\cdot\Sigma_1\cdot\Sigma_2} e_1 : \tau' \qquad \Lambda; [x : \forall\Sigma_2.\tau'] \vdash_{(\Sigma+c)\cdot\Sigma_1} e_2 : \tau$$

$$Weaker\,((\Sigma + c) \cdot \Sigma_1 \downarrow (FTV(\tau') \cap dom\,(\Sigma \cdot \Sigma_1)) \setminus FTV(\Lambda), 0).$$

Then Theorem 1 shows there would exist $\Sigma'$ and $\Lambda'$ such that $\Sigma' \leq_1 \Sigma$, and

$$\Lambda \cdot \Lambda'; [\,] \vdash_{(\Sigma'+c)\cdot\Sigma_1\cdot\Sigma_2} a_1 : \tau' \qquad \vdash_{\Sigma'} \sigma_1 : \Lambda \cdot \Lambda' \qquad Crit\,(\Sigma' + c \downarrow FTV(\Lambda')).$$

https://doi.org/10.1017/S0956796800001593 Published online by Cambridge University Press

Define $\Sigma''$ such that for all $\alpha$ in $dom(\Sigma'') = dom(\Sigma')$,

$$\Sigma''(\alpha) = \begin{cases} \min(\Sigma'(\alpha), -c) & \text{if } \alpha \text{ is in } FTV(\Lambda), \\ \Sigma'(\alpha) & \text{otherwise,} \end{cases}$$

so that $\Sigma'' \leq_1 \Sigma'$, and $Weaker((\Sigma'' + c) \cdot \Sigma_1 \downarrow FTV(\Lambda \cdot \Lambda_1, \tau') \cap dom(\Sigma'' \cdot \Sigma_1), 0)$. Then by Lemma 8, there would exist a $\Sigma'''$ such that $\Sigma''' + c \leq_1 \Sigma'' + c$ (and thus $\Sigma''' \leq_1 \Sigma''$), and

$$\Lambda \cdot \Lambda'; [\,] \vdash_{(\Sigma''' + c) \cdot \Sigma_1} [a_1/x]e_2 : \tau.$$

But since $\vdash_{\Sigma'''} \sigma_1 : \Lambda \cdot \Lambda'$ and the fact that $Crit(\Sigma''' \downarrow \Lambda \cdot \Lambda')$, induction on the evaluation of the substitution contradicts this typing, so the assumption of $e$ being typable is false. □

## Acknowledgments

Many thanks go to Robert Harper, Mark Lillibridge and the two anonymous referees for their many helpful ideas. I would also like to thank Mark Leone, Philip Wadler and Peter Lee for their comments.

## References

Damas, L. and Milner, R. (1982) *Principal type-schemes for functional programs.* In *9th ACM Symposium on Principles of Programming Languages*, pp. 207–212. ACM Press.

Damas, L. (1985) *Type Assignment in Programming Languages.* PhD Thesis, University of Edinburgh.

Harper, R. (1993) A Simplified Account of Polymorphic References. *Technical Report CMU-CS-93-169*, Carnegie Mellon University.

Hindley, J. R. (1969) The principal type scheme of an object in combinatory logic. *Trans. Am. Math. Soc.*, **146**: 29–60.

Hindley, J. R. and Seldin, J. P. (1986) *Introduction to Combinators and λ-Calculus.* London Mathematical Society Student Texts, **1**. Cambridge University Press.

Hoang, M., Mitchell, J. and Viswanathan, R. (1993) Standard ML weak polymorphism and imperative constructs. In *8th Symposium on Logic in Computer Science*, pp. 15–25. IEEE Press.

Leroy, X. (1992) Polymorphic Typing of An Algorithmic Language. *Technical Report 1778*, Institute National de Recherche en Informatique et en Automatique. (English translation of PhD Thesis, Université Paris VII.)

Leroy, X. (1993) Polymorphism by name for references and continuations. In *20th ACM Symposium on Principles of Programming Languages*, pp. 220–231. ACM Press

Leroy, X. and Weis, P. (1991) Polymorphic type inference and assignment. In *18th ACM Symposium on Principles of Programming Languages*, pp. 291–302. ACM Press.

MacQueen, D. (1992) *Source code for SML/NJ type inference algorithm*, Versions 0.66, 0.75, and 0.93.

Milner, R. (1978) A theory of type polymorphism in programming languages. *J. Computer & System Sci.*, **17**: 348–375.

Milner, R., Tofte, M. and Harper, R. (1990) *The Definition of Standard ML.* MIT Press.

O'Toole, Jr., J. W. (1989) Type Abstraction Rules for References: A Comparison of Four Which Have Achieved Notoriety. *Technical report MIT–LCS–TM–390*, Massachussets Institute of Technology.

Reynolds, J. C. (1989) Syntactic Control of Interference, Part 2. In *16th Int. Colloquium on Automata, Languages and Programming*, pp. 704–722. Springer-Verlag.

Talpin, J.-P. and Jouvelot, P. (1992) The type and effect discipline. In *7th IEEE Symposium on Logic in Computer Science*, pp. 162–173. IEEE Press.

Talpin, J.-P. and Jouvelot, P. (1992) Polymorphic type, region and effect inference. *J. Functional Programming*, **2**(3): 245–271.

Tofte, M. (1988) *Operational Semantics and Polymorphic Type Inference*. PhD Thesis, University of Edinburgh.

Tofte, M. (1990) Type inference for polymorphic references. *Infor. & Computation*, **89**: 1–34.

Wright, A. K. (1992) Typing references by effect inference. In *4th Euro. Symposium on Programming: Lecture Notes in Computer Science 582*, pp. 473–491. Springer-Verlag.

Wright, A. K. (1993) Polymorphism for Imperative Languages without Imperative Types. *Technical Report 93–200*, Rice University.

Wright, A. K. and Felleisen, M. (1991) A Syntactic Approach to Type Soundness. *Technical Report 91–160*, Rice University.