Type Assignment for Intersections and Unions in Call-by-Value Languages

Joshua Dunfield and Frank Pfenning

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
{joshuad,fp}@cs.cmu.edu

Abstract. We develop a system of type assignment with intersection types, union types, indexed types, and universal and existential dependent types that is sound in a call-by-value functional language. The combination of logical and computational principles underlying our formulation naturally leads to the central idea of type-checking subterms in evaluation order. We thereby provide a uniform generalization and explanation of several earlier isolated systems. The proof of progress and type preservation, usually formulated for closed terms only, relies on a notion of definite substitution.

1 Introduction

Conventional static type systems are tied directly to the expression constructs available in a language. For example, functions are classified by function types $A \to B$, pairs are classified by product types $A * B$, and so forth. In more advanced type systems we find type constructs that are independent of any particular expression construct. The best-known examples are parametric polymorphism $\forall t.A$ and intersection polymorphism $A \wedge B$. Such types can be seen as expressing more complex properties of programs. For example, if we read the judgment $e : A$ as *e satisfies property A*, then $e : A \wedge B$ expresses that e satisfies both property A and property B. We call such types *property types*. Our long-term goal is to integrate a rich system of property types into practical languages such as Standard ML [9], in order to express and verify detailed invariants of programs as part of type-checking.

In this paper we design a system of property types specifically for call-by-value languages. We show that the resulting system is type-safe, that is, satisfies the type preservation and progress theorems. We include indexed types $\delta(i)$, intersection types $A \wedge B$, a greatest type $\top$, universal dependent types $\Pi a{:}\gamma.\, A$, union types $A \vee B$, an empty type $\bot$, and existential dependent types $\Sigma a{:}\gamma.\, A$. We thereby combine, unify, and extend prior work on intersection types [6], union types [11, 2] and dependent types [17].

To appear at FOSSACS'03, April 2003, Warsaw, Poland.

Several principles emerge from our investigation. Perhaps most important is that type assignment may visit subterms in evaluation order, rather than just relying on immediate subterms. We also confirm the critical importance of a logically motivated design for subtyping and type assignment. The resulting orthogonality of various property type constructs greatly simplifies the theory and allows one to understand each concept in isolation. As a consequence, simple types, intersection types [6], and indexed and dependent types [17] are extended *conservatively.* There are also interesting technical aspects in our proof of progress and preservation: Usually these can be formulated entirely for closed expressions; here we needed to generalize the properties by allowing so-called *definite substitutions.* Our type system is designed to allow effects (in particular, mutable references), but in order to concentrate on more basic issues, we do not include them explicitly in this paper (see [6] for the applicable techniques to handle mutable references).

The results in this paper constitute the first step towards a practical type system. The system of pure type assignment presented here is undecidable; to remedy this, we have formulated another version based on bidirectional type-checking (in the style of [6, 7]) of programs containing some type annotations, where we variously *check* an expression against a type or else *synthesize* the expression's type. However, we do not yet have a formal proof of decidability, nor any significant experience with checking realistic programs. Our confidence in the practicality of the system rests on prior work on intersection and dependent types in isolation.

The remainder of the paper is organized as follows. We start by defining a small and conventional functional language with subtyping, in a standard call-by-value semantics. We then add several forms of property types: intersection types, indexed types, and universal dependent types. As we do so, we motivate our typing and subtyping rules through examples, showing how our particular formulation arises out of our demand that the theorems of *type preservation* and *progress* hold. Then we add the *indefinite* property types: the empty type $\bot$, the union type $\vee$, and the existential dependent type Σ. To be sound, these must visit subterms in evaluation order. After proving some novel properties of judgments and substitutions, we prove preservation and progress. Finally, we discuss related work and conclude.

2 The Base Language

We start by defining a standard call-by-value functional language (Figure 1) with functions, a unit type (used in a few examples), and recursion, to which we will add various constructs and types. Expressions do not contain types, because we are formulating a pure type assignment system. We distinguish between variables x that stand for values and variables f that stand for expressions, where the fs arise only from fixed points. The form of the typing judgment is $\Gamma \vdash e : A$ where Γ is a context typing variables x and f. The typing rules here are standard (Figure 2); the subsumption rule utilizes a subtyping judgment $\Gamma \vdash A \leq B$

$$A, B, C, D ::= \mathbf{1} \mid A \to B$$
$$e ::= x \mid f \mid () \mid \lambda x.\, e \mid e_1(e_2) \mid \mathbf{fix}\ f.\, e$$

Fig. 1. Syntax of types and terms in the initial language

$$\frac{\Gamma \vdash B_1 \leq A_1 \quad \Gamma \vdash A_2 \leq B_2}{\Gamma \vdash A_1 \to A_2 \leq B_1 \to B_2}\ (\to) \qquad \frac{}{\Gamma \vdash \mathbf{1} \leq \mathbf{1}}\ (\mathbf{1})$$

$$\frac{\Gamma(x) = A}{\Gamma \vdash x : A}\ (\text{var}) \qquad \frac{\Gamma(f) = A}{\Gamma \vdash f : A}\ (\text{fixvar}) \qquad \frac{\Gamma \vdash e : A \quad \Gamma \vdash A \leq B}{\Gamma \vdash e : B}\ (\text{sub})$$

$$\frac{\Gamma \vdash e_1 : A \to B \quad \Gamma \vdash e_2 : A}{\Gamma \vdash e_1(e_2) : B}\ (\to E) \qquad \frac{\Gamma, f{:}A \vdash e : A}{\Gamma \vdash \mathbf{fix}\ f.\, e : A}\ (\mathbf{fix})$$

$$\frac{\Gamma, x{:}A \vdash e : B}{\Gamma \vdash \lambda x.\, e : A \to B}\ (\to I) \qquad \frac{}{\Gamma \vdash () : \mathbf{1}}\ (\mathbf{1}I)$$

Fig. 2. Subtyping and typing in the initial language

meaning that A is a subtype of B in context Γ. The interpretation is that the set of values of type A is a subset of the set of values of type B. The context Γ is not used in the subtyping rules of Figure 2, but we subsequently augment the subtyping system with rules that refer to Γ. The rule $(\to)$ is the standard subtyping rule for function types, contravariant in the argument and covariant in the result; $(\mathbf{1})$ is obvious. It is easy to prove that subtyping is decidable, reflexive ($\Gamma \vdash A \leq A$), and transitive (if $\Gamma \vdash A \leq B$ and $\Gamma \vdash B \leq C$ then $\Gamma \vdash A \leq C$); as we add rules to the subtyping system, we maintain these properties.

A call-by-value operational semantics defining a relation $e \mapsto e'$ is given in Figure 3. We use v for values, and write e value if e is a value. We write E for an evaluation context—a term containing a hole $[]$; $E[e']$ denotes E with its hole replaced by e'.

3 Definite Property Types

Definite types accumulate positive information about expressions. For instance, the intersection type $A \wedge B$ expresses the conjunction of the properties A and B. We later introduce *indefinite types* such as $A \vee B$ which encompass expressions that have either property A or property B, although it is unknown which one.

3.1 Refined Datatypes

We now add datatypes with refinements (Figure 4). $c(e)$ denotes a datatype constructor c applied to an argument e; the destructor **case** e **of** ms denotes a case over e with one layer of non-redundant and exhaustive matches ms. We also add pairs so a constructor can take exactly one argument, but elide the straightforward syntax and rules. Each datatype is *refined*, in the manner of [5], by an *atomic subtyping* relation $\preceq$ over *datasorts* δ. Each datasort identifies a

$$\text{Values} \quad v ::= x \mid () \mid \lambda x.\, e$$
$$\text{Evaluation contexts} \quad E ::= [] \mid E(e) \mid v(E)$$

$$\frac{e' \;\mapsto_R\; e''}{E[e'] \;\mapsto\; E[e'']}\;(\text{ev-context}) \qquad \begin{array}{rcl} (\lambda x.\, e)\, v & \mapsto_R & [v/x]\, e \\ \mathbf{fix}\; f.\, e & \mapsto_R & [\mathbf{fix}\; f.\, e \,/\, f]\, e \end{array}$$

Fig. 3. A small-step call-by-value semantics

$$\begin{array}{rcl} ms & ::= & \cdot \mid c(x) \Rightarrow e \,|\, ms \\ e & ::= & \ldots \mid c(e) \mid \mathbf{case}\; e\; \mathbf{of}\; ms \\ v & ::= & \ldots \mid c(v) \\ E & ::= & \ldots \mid c(E) \mid \mathbf{case}\; E\; \mathbf{of}\; ms \end{array}$$

Fig. 4. Extending the language with datatypes

subset of values of the form $c(v)$, yielding definite information about a value. For example, datasorts true and false identify singleton subsets of values of the type bool.

A new subtyping rule defines subtyping for datasorts in terms of the atomic subtyping relation $\preceq$:

$$\frac{\delta_1 \preceq \delta_2}{\Gamma \vdash \delta_1 \leq \delta_2}\;(\delta)$$

To maintain reflexivity and transitivity of subtyping, we require the same properties of atomic subtyping: $\preceq$ must be reflexive and transitive.

Since we will subsequently further refine our datatypes by indices, we defer discussion of the typing rules.

3.2 Intersections

The typing $e : A \wedge B$ expresses that e has type A and type B. The subtyping rules for $\wedge$ capture this:

$$\frac{\Gamma \vdash A \leq B_1 \quad \Gamma \vdash A \leq B_2}{\Gamma \vdash A \leq B_1 \wedge B_2}\;(\wedge\text{R}) \qquad \frac{\Gamma \vdash A_1 \leq B}{\Gamma \vdash A_1 \wedge A_2 \leq B}\;(\wedge\text{L}_1) \qquad \frac{\Gamma \vdash A_2 \leq B}{\Gamma \vdash A_1 \wedge A_2 \leq B}\;(\wedge\text{L}_2)$$

We omit the common distributivity rule

$$\frac{}{(A \to B) \wedge (A \to B') \leq A \to (B \wedge B')}$$

which Davies and Pfenning showed to be unsound in the presence of mutable references [6]. Moreover, without the above rule, no subtyping rule contains more than one type constructor: the rules are orthogonal. As we add type constructors and subtyping rules, we will maintain this orthogonality.

On the level of typing, we can introduce an intersection with the rule

$$\frac{\Gamma \vdash v : A_1 \quad \Gamma \vdash v : A_2}{\Gamma \vdash v : A_1 \wedge A_2}\;(\wedge I)$$

$$P \;::=\; \bot \mid i \doteq j \mid \ldots \qquad\qquad \overline{\cdot} = \cdot$$

$$\Gamma \;::=\; \cdot \mid \Gamma, x{:}A \mid \Gamma, a{:}\gamma \mid \Gamma, P \qquad\qquad \begin{aligned} \overline{\Gamma, x{:}A} &= \overline{\Gamma} \\ \overline{\Gamma, a{:}\gamma} &= \overline{\Gamma}, a{:}\gamma \\ \overline{\Gamma, P} &= \overline{\Gamma}, P \end{aligned}$$

Fig. 5. Propositions P, contexts Γ, and the restriction function $\overline{\Gamma}$

and eliminate it with

$$\frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash e : A_1}\,(\wedge E_1) \qquad \frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash e : A_2}\,(\wedge E_2)$$

Note that $(\wedge I)$ can only type values v, not arbitrary expressions, following Davies and Pfenning [6] who showed that in the presence of mutable references, allowing non-values destroys type preservation.

The $\wedge$-elimination rules are derivable via (sub) with the ($\wedge$L1) and ($\wedge$L2) subtyping rules. However, we include them because they are not derivable in a bidirectional system such as that of [7].

3.3 Greatest Type: $\top$

It is easy to incorporate a greatest type $\top$, which can be thought of as the 0-ary form of $\wedge$. The rules are simply

$$\frac{}{\Gamma \vdash A \leq \top}\,(\top\mathrm{R}) \qquad \frac{}{\Gamma \vdash v : \top}\,(\top I)$$

There is no left subtyping rule. The typing rule is essentially the 0-ary version of $(\wedge I)$, the rule for binary intersection. If we allow $(\top I)$ to type non-values, the progress theorem fails: $\vdash ()\,() : \top$, but $()\,()$ is neither a value nor a redex.

3.4 Index Refinements and Universal Dependent Types Π

Now we add index refinements, which are dependent types over a restricted domain, closely following Xi and Pfenning [17], Xi [15,16], and Dunfield [7]. This refines datatypes not only by datasorts, but by indices drawn from some constraint domain: the type $\delta(i)$ is the refinement by δ and index i.

To accommodate index refinements, several changes must be made to the systems we have constructed so far. The most drastic is that Γ can include *index variables* a, b and propositions P as well as program variables. Because the program variables are irrelevant to the index domain, we can define a *restriction function* $\overline{\Gamma}$ that yields its argument Γ without program variable typings (Figure 5). No variable may appear twice in Γ, but ordering of the variables is now significant because of dependencies.

Our formulation, like Xi's, requires only a few properties of the constraint domain: There must be a way to decide a consequence relation $\overline{\Gamma} \models P$ whose interpretation is that given the index variable typings and propositions in $\overline{\Gamma}$, the proposition P must hold. There must be a relation $i \doteq j$ denoting index equality. There must be a way to decide a relation $\overline{\Gamma} \vdash i : \gamma$ whose interpretation is that i has *sort* γ in $\overline{\Gamma}$. Note the stratification: terms have types, indices have sorts; terms and indices are distinct. Our proofs require that $\models$ be a consequence relation, that $\doteq$ be an equivalence relation, that $\cdot \not\models \bot$, and that both $\models$ and $\vdash$ have the obvious substitution and weakening properties; see [7] for details.

Each datatype has an associated atomic subtyping relation on datasorts, and an associated sort whose indices refine the datatype. In our examples, we work in a domain of integers $\mathcal{N}$ with $\doteq$ and some standard arithmetic operations ($+$, $-$, $*$, $<$, and so on); each datatype is refined by indices of sort $\mathcal{N}$. Then $\overline{\Gamma} \models P$ is decidable provided the inequalities in P are linear.

We add an infinitary definite type $\Pi a{:}\gamma.\, A$, introducing an index variable a universally quantified over indices of sort γ. One can also view Π as a dependent product restricted to indices (instead of arbitrary terms).

Example. Assume we define an datatype of integer lists: a list is either $\mathsf{Nil()}$ or $\mathsf{Cons}(h, t)$ for some integer h and list t. Refine this type by a datasort odd if the list's length is odd, even if it is even. We also refine the lists by their length, so Nil has type $\mathbf{1} \to \mathsf{even}(0)$, and Cons has type $(\Pi a{:}\mathcal{N}.\, \mathsf{int} * \mathsf{even}(a) \to \mathsf{odd}(a+1)) \wedge (\Pi a{:}\mathcal{N}.\, \mathsf{int} * \mathsf{odd}(a) \to \mathsf{even}(a+1))$. Then the function

$$\mathbf{fix}\ \mathit{repeat}.\, \lambda x.\, \mathbf{case}\ x\ \mathbf{of}\ \mathsf{Nil} \Rightarrow \mathsf{Nil}\ \mid \mathsf{Cons}(h, t) \Rightarrow \mathsf{Cons}(h, \mathsf{Cons}(h, \mathit{repeat}(t)))$$

has type $\Pi a{:}\mathcal{N}.\, \mathsf{list}(a) \to \mathsf{even}(2 * a)$.

To handle the indices, we modify the subtyping rule δ from Section 3.1 so that it checks (separately) the datasorts δ_1, δ_2 and the indices i, j:

$$\frac{\delta_1 \preceq \delta_2 \quad \overline{\Gamma} \vdash i \doteq j}{\Gamma \vdash \delta_1(i) \leq \delta_2(j)}\ (\delta)$$

We assume the constructors c are typed by a judgment $\overline{\Gamma} \vdash c : A \to \delta(i)$ where A is any type and $\delta(i)$ is some refined type. The typing $A \to \delta(i)$ need not be unique; indeed, a constructor should often have more than one refined type. The rule for constructor application is

$$\frac{\overline{\Gamma} \vdash c : A \to \delta(i) \quad \Gamma \vdash e : A}{\Gamma \vdash c(e) : \delta(i)}\ (\delta I)$$

To type $\mathbf{case}\ e\ \mathbf{of}\ ms$, we check that all the matches in ms have the same type, under a context appropriate to each arm; this is how propositions P arise. The context Γ may be contradictory ($\overline{\Gamma} \models \bot$) if the case arm can be shown to be unreachable by virtue of the index refinements of the constructor type and the value cased upon. In order to not typecheck unreachable arms, we have

$$\frac{\overline{\Gamma} \models \bot}{\Gamma \vdash e : A}\ (\text{contra})$$

We also do not check case arms that are unreachable by virtue of the *datasort* refinements. For a complete accounting of constructor typing and the rules for typing **case** expressions, see [7].

The subtyping rules for Π are

$$\frac{\Gamma \vdash [i/a]A \leq B \quad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash \Pi a{:}\gamma.\, A \leq B}\,(\Pi\mathrm{L}) \qquad \frac{\Gamma, b{:}\gamma \vdash A \leq B}{\Gamma \vdash A \leq \Pi b{:}\gamma.\, B}\,(\Pi\mathrm{R})$$

The left rule allows one to instantiate a quantified index variable a to an index i of appropriate sort. The right rule states that if $A \leq B$ regardless of an index variable b, A is also a subtype of $\Pi b{:}\gamma.\, B$. Of course, b cannot occur free in A.

The typing rules for Π are

$$\frac{\Gamma, a{:}\gamma \vdash v : A}{\Gamma \vdash v : \Pi a{:}\gamma.\, A}\,(\Pi I) \qquad \frac{\Gamma \vdash e : \Pi a{:}\gamma.\, A \quad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash e : [i/a]\, A}\,(\Pi E)$$

Like $(\wedge I)$, and for similar reasons (to maintain type preservation), (ΠI) is restricted to values. Moreover, if γ is an empty sort, progress would fail if the rule were not thus restricted.

4 Indefinite Property Types

We now have a system with definite types $\wedge$, $\top$, Π; see [7] for a detailed account of this system and its bidirectional version. The typing and subtyping rules are both orthogonal and internally regular: no rule mentions both $\top$ and $\wedge$, $(\top I)$ is a 0-ary version of $(\wedge I)$, and so on. However, one cannot express the types of functions with indeterminate result type. A simple example is a `filter` f l function on lists of integers, which returns the elements of l for which f returns true. It has the ordinary type `filter` : $(\mathsf{int}\to\mathsf{bool}) \to \mathsf{list} \to \mathsf{list}$. Indexing lists by their length, the refined type should look like

$$\texttt{filter} : \Pi n{:}\mathcal{N}.\, (\mathsf{int}\to\mathsf{bool}) \to \mathsf{list}(n) \to \mathsf{list}(\_\_)$$

But we cannot fill in the blank. Xi's solution [17, 15] was to add dependent sums $\Sigma a{:}\gamma.\, A$ quantifying existentially over index variables. Then we can express the fact that `filter` returns a list of some indefinite length m as follows[1]:

$$\texttt{filter} : \Pi n{:}\mathcal{N}.\, (\mathsf{int}\to\mathsf{bool}) \to \mathsf{list}(n) \to (\Sigma m{:}\mathcal{N}.\, \mathsf{list}(m))$$

For similar reasons, we also occasionally need 0-ary and binary indefinite types—the empty type and union types, respectively. We begin with the binary case.

4.1 Unions

On values, the binary indefinite type should simply be a union in the ordinary sense: if $\vdash v : A \vee B$ then either $\vdash v : A$ or $\vdash v : B$. This leads to the following subtyping rules which are dual to the intersection rules.

$$\frac{\Gamma \vdash A_1 \leq B \quad \Gamma \vdash A_2 \leq B}{\Gamma \vdash A_1 \vee A_2 \leq B}\,(\vee\mathrm{L}) \quad \frac{\Gamma \vdash A \leq B_1}{\Gamma \vdash A \leq B_1 \vee B_2}\,(\vee\mathrm{R}_1) \quad \frac{\Gamma \vdash A \leq B_2}{\Gamma \vdash A \leq B_1 \vee B_2}\,(\vee\mathrm{R}_2)$$

[1] The additional constraint $m \leq n$ can be expressed by a *subset sort*; see Xi [16, 15].

The introduction rules directly express the simple logical interpretation:

$$\frac{\Gamma \vdash e : A}{\Gamma \vdash e : A \vee B}\ (\vee I_1) \qquad \frac{\Gamma \vdash e : B}{\Gamma \vdash e : A \vee B}\ (\vee I_2)$$

The elimination rule is harder to formulate. It is clear that if $e : A \vee B$ and e evaluates to a value v, then either $v : A$ or $v : B$. So we should be able to reason by cases, similar to the usual disjunction elimination rule in natural deduction. However, there are several complications. The first is that $A \vee B$ is a property type. That is, we cannot have a `case` construct in the ordinary sense since the members of the union are not tagged.[2]

As a simple example, consider

$$\begin{array}{l} f : (B \to D) \wedge (C \to D) \\ g : A \to (B \vee C) \\ x : A \end{array}$$

Then $f(g(x))$ should be type correct and have type D. At first this might seem doubtful, because the type of f does not directly show how to treat an argument of type $B \vee C$. However, whatever g returns must be a closed value v, and must therefore either have type B or type C. In both cases $f(v)$ should be well-typed and return a result of type D.

Note that we can distinguish cases on the result of $g(x)$ because it is evaluated *before* f is called.[3] In general, we allow case distinction on the type of the next expression to be evaluated. This guarantees both progress and preservation. The rule is then

$$\frac{\Gamma \vdash e' : A \vee B \quad \begin{array}{c}\Gamma, x{:}A \vdash E[x] : C \\ \Gamma, y{:}B \vdash E[y] : C\end{array}}{\Gamma \vdash E[e'] : C}\ (\vee E)$$

The use of the evaluation context $E[\,]$ guarantees that e' is the next expression to be evaluated, following our informal reasoning above. In the example, $e' = g(x)$ and $E[\,] = f[\,]$.

Several generalizations of this rule come to mind that are in fact unsound in our setting. For example, allowing simultaneous abstraction over several occurrences of e', as in a rule proposed in [2],

$$\frac{\Gamma \vdash e' : A \vee B \quad \begin{array}{c}\Gamma, x{:}A \vdash e : C \\ \Gamma, x{:}B \vdash e : C\end{array}}{\Gamma \vdash [e'/x]\, e : C}\ (\vee E')$$

is unsound here: two occurrences of the identical e' could return different results (the first of type A, the second of type B), while the rule above assumes consistency. Similarly, we cannot allow the occurrence of e' to be in a position where

[2] Pierce's `case` [11] is a syntactic marker for where to apply the elimination rule. Clearly, a pure type assignment system should avoid this. It appears we can avoid it even in a bidirectional system; further discussion is beyond the scope of this paper.

[3] If arguments were passed by name instead of by value, this would be unsound in a language with effects: evaluation of the same expression $e : A \vee B$ could sometimes return a value of type A and sometimes a value of type B.

it might not be evaluated. That is, in $(\vee E')$ it is not enough to require that there be exactly one occurrence of x in e, because, for example, if we consider the context

$$\begin{array}{l} f : ((1{\to}B) \to D) \wedge ((1{\to}C) \to D), \\ g : A \to (B \vee C), \\ x : A \end{array}$$

and term $f\,(\lambda y.\, g(x))$, then f may use its argument at multiple types, eventually evaluating $g(x)$ multiple times with different possible answers. Thus, treating it as if all occurrences must all have type B or all have type C is unsound. If we restrict the rule so that e' must be a value, as in [13], we obtain a sound but impractical rule—a typechecker would have to guess e', and if it occurs more than once, a subset of its occurrences.

A final generalization suggests itself: we might allow the subterm e' to occur exactly once, and in any position where it would definitely have to be evaluated exactly once for the whole expression to be evaluated. Besides the difficulty of characterizing such positions, even this apparently innocuous generalization is unsound for the empty type $\bot$.

4.2 The Empty Type

The 0-ary indefinite type is the empty or void type $\bot$; it has no values. For $\top$ we had one right subtyping rule; for $\bot$, following the principle of duality, we have one left rule:

$$\frac{}{\Gamma \vdash \bot \leq A}\ (\bot\mathrm{L})$$

For example, the term $\omega = (\mathbf{fix}\ f.\,\lambda x.\, f(x))\,()$ has type $\bot$. For an elimination rule $(\bot E)$, we can proceed by analogy with $(\vee E)$:

$$\frac{\Gamma \vdash e' : \bot}{\Gamma \vdash E[e'] : C}\ (\bot E)$$

As before, the expression typed must be an evaluation context E with redex e'. Viewing $\bot$ as a 0-ary union, we had two additional premises in $(\vee E)$, so we have none now. $(\bot E)$ is sound, but the generalization mentioned at the end of the previous section violates progress (Theorem 3). This is easy to see through the counterexample $(()\,())(\omega)$.

4.3 Existential Dependent Types: Σ

Now we add an infinitary indefinite type Σ. Just as we have come to expect, the subtyping rules are dual to the rules for the corresponding definite type (in this case Π):

$$\frac{\Gamma, a{:}\gamma \vdash A \leq B}{\Gamma \vdash \Sigma a{:}\gamma.\, A \leq B}\ (\Sigma\mathrm{L}) \qquad \frac{\Gamma \vdash A \leq [i/b]\,B \quad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash A \leq \Sigma b{:}\gamma.\, B}\ (\Sigma\mathrm{R})$$

The typing rule that introduces Σ is simply

$$\frac{\Gamma \vdash e : [i/a]\,A \quad \overline{\Gamma} \vdash i : \gamma}{\Gamma \vdash e : \Sigma a{:}\gamma.\,A}\ (\Sigma I)$$

For the elimination rule, we continue with a restriction to evaluation contexts:

$$\frac{\Gamma \vdash e' : \Sigma a{:}\gamma.\,A \quad \Gamma, a{:}\gamma, x{:}A \vdash E[x] : C}{\Gamma \vdash E[e'] : C}\ (\Sigma E)$$

Not only is the restriction consistent with the elimination rules for $\bot$ and $\vee$, but it is required. The counterexample for $\bot$ suffices: Suppose that the rule were unrestricted, so that it typed any e containing some subterm e'. Let $e' = \omega$ and $e = (()\,())(\omega)$. Since $e' : \bot$, by subsumption e' has type $\Sigma a{:}\bot.\,A$ for any A, and by the (contra) rule, $a{:}\bot, x{:}A \vdash (()\,())x : C$ (where $\bot$ is the empty sort). Now we can apply the unrestricted rule to conclude $\vdash (()\,())e' : C$ for any C, contrary to progress.

4.4 Type-Checking in Evaluation Order

The following rule internalizes a kind of substitution principle for evaluation contexts and allows us to type-check a term in evaluation order.

$$\frac{\Gamma \vdash e' : A \quad \Gamma, x{:}A \vdash E[x] : C}{\Gamma \vdash E[e'] : C}\ (\text{direct})$$

Perhaps surprisingly, this rule is not only admissible but derivable in our system: from $e' : A$ we can conclude $e' : A \vee A$ and then apply ($\vee E$). However, the corresponding bidirectional rule is not admissible, and so must be primitive in a bidirectional system [7].

Thus, in either the type assignment or bidirectional systems, we can choose to type-check the term in evaluation order. This has a clear parallel in Xi's work [15], which is bidirectional and contains both Π and Σ. There, the order in which terms are typed is traditional, not guided by evaluation order. However, Xi's elaboration algorithm in the presence of Π and Σ transforms the term into a let-normal form, which has a similar effect.

5 Properties of Subtyping

The rules of subtyping were formulated so that the premises are always smaller than the conclusion. Since we assume that $\models$ and $\vdash$ in the constraint domain are decidable, we obtain decidability immediately.

Theorem 1. *$\Gamma \vdash A \leq B$ is decidable.*

We omitted rules for reflexivity and transitivity of subtyping without loss of expressive power, because they are admissible.

Lemma 1 (Reflexivity and Transitivity of $\leq$). *For any context Γ, $\Gamma \vdash A \leq A$. If $\Gamma \vdash A \leq B$ and $\Gamma \vdash B \leq C$ then $\Gamma \vdash A \leq C$.*

$$\frac{}{\Gamma' \vdash \cdot : \cdot}\ (\text{empty-}\sigma) \qquad \frac{\Gamma' \vdash \sigma : \Gamma \quad \overline{\Gamma'} \models [\sigma]P}{\Gamma' \vdash \sigma : \Gamma, P}\ (\text{prop-}\sigma)$$

$$\frac{\Gamma' \vdash \sigma : \Gamma \quad \overline{\Gamma'} \vdash i : \gamma}{\Gamma' \vdash \sigma, i/a : \Gamma, a{:}\gamma}\ (\text{ivar-}\sigma) \qquad \frac{\Gamma' \vdash \sigma : \Gamma \quad \Gamma' \vdash v : [\sigma]A}{\Gamma' \vdash \sigma, v/x : \Gamma, x{:}A}\ (\text{pvar-}\sigma)$$

Fig. 6. Substitution typing

Proof. For reflexivity, by induction on A. For transitivity, by induction on the derivations; in each case at least one derivation becomes smaller. In the cases $(\Sigma\mathrm{R})/(\Sigma\mathrm{L})$ and $(\Pi\mathrm{R})/(\Pi\mathrm{L})$ we substitute an index i for a parameter a in a derivation. □

In addition we have a large set of inversion properties, which are purely syntactic in our system. We elide the lengthy statement of these properties here.

6 Properties of Values

For the proof of type safety, we need a key property: *values are always definite.* That is, once we obtain a value v, even though v might have type $A \vee B$, it *must* be possible to assign a definite type to v. In order to make this precise, we formulate substitutions σ that substitute values and indices, respectively, for several program variables x and index variables a. First we prove a simple lemma relating values and evaluation contexts.

Lemma 2 (Value monotonicity). *If $E[e']$ value then: (1) e' value; (2) for any v value, $E[v]$ value.*

Proof. By structural induction on E. □

6.1 Substitutions

Figure 6 defines a typing judgment for substitutions $\Gamma' \vdash \sigma : \Gamma$. It could be more general; here we are only interested in substitutions of values for program variables and indices for index variables that verify the logical assumptions of the constraint domain. Note in particular that substitutions σ do *not* substitute for fixed point variables. Application of a substitution σ to a term e or type A, is in the usual (capture-avoiding) manner.

Lemma 3 (Substitution).

(i) *If $\Gamma \vdash A \leq B$ and $\Gamma' \vdash \sigma : \Gamma$, then $\Gamma' \vdash [\sigma]A \leq [\sigma]B$.*
(ii) *If $\mathcal{D}$ derives $\Gamma \vdash e : A$ and $\Gamma' \vdash \sigma : \Gamma$, then there exists $\mathcal{D}'$ deriving $\Gamma' \vdash [\sigma]e : [\sigma]A$. Moreover, if $\overline{\Gamma} = \Gamma$ (that is, Γ contains only index variables and index constraints), there is such a $\mathcal{D}'$ not larger than $\mathcal{D}$ (that is, the number of typing rules used in $\mathcal{D}'$ is at most the number used in $\mathcal{D}$).*
(iii) *If $\Gamma \vdash e : B$ and $\Gamma, f{:}B \vdash e' : A$ then $\Gamma \vdash [e/f]\, e' : A$.*

Similar properties hold for matches ms.

Proof. By induction on the respective derivations. □

6.2 Definiteness

We formalize definiteness as follows, for typing judgments with (possibly) non-empty contexts:

Definition 1. *A typing judgment $\Gamma \vdash e : A$ is definite with respect to a substitution $\vdash \sigma : \Gamma$ if and only if*

(i) $\vdash [\sigma]A \leq \bot$ *is not derivable;*
(ii) if $\vdash [\sigma]A \leq B_1 \vee B_2$ *then* $\vdash [\sigma]e : B_1$ *or* $\vdash [\sigma]e : B_2$*;*
(iii) if $\vdash [\sigma]A \leq \Sigma b{:}\gamma.\, B$*, there exists an index* $\vdash i : \gamma$ *where* $\vdash [\sigma]e : [i/b]\, B$*.*

Definition 2. *A substitution $\vdash \sigma : \Gamma$ is definite iff for all A such that $\Gamma(x) = A$, $\Gamma \vdash x : A$ is definite with respect to σ.*

To prove value definiteness (Theorem 2), which is central in the proof of type safety, we first prove a weaker form of the theorem that depends on the definiteness of the substitution σ. This weak form will allow us to show that *every* well-typed substitution is definite, which will lead directly to a proof of the theorem.

Lemma 4 (Weak value definiteness). *If $\Gamma \vdash v : A$ where σ is a definite substitution and $\vdash \sigma : \Gamma$, then $\Gamma \vdash v : A$ is definite with respect to σ, that is:*

(i) it is not the case that $\Gamma \vdash A \leq \bot$*;*
(ii) if $\Gamma \vdash A \leq A_1 \vee A_2$ *then* $\vdash [\sigma]v : [\sigma]A_1$ *or* $\vdash [\sigma]v : [\sigma]A_2$*;*
(iii) if $\Gamma \vdash A \leq \Sigma b{:}\gamma.\, B$ *then there exists* $\vdash i : \gamma$ *where* $\vdash [\sigma]v : [\sigma, i/b]\, B$*.*

Proof. By induction on the derivation of $\Gamma \vdash v : A$. The term v is a value, so we need not consider rules that cannot type values. Furthermore, the (contra) case cannot arise. Most cases follow easily from the IH and properties of subtyping (reflexivity, transitivity, inversion). For (var) we use the fact that σ is a definite substitution. That leaves only the contextual rules, $(\bot E)$, $(\vee E)$, (direct) and (ΣE); we show the first two cases (the last two are similar to $(\vee E)$):

- **Case** $(\bot E)$: $\mathcal{D} = \dfrac{\Gamma \vdash e' : \bot}{\Gamma \vdash E[e'] : A}\ (\bot E)$

  $E[e']$ is a value. By Lemma 2, e' is a value. $\Gamma \vdash \bot \leq \bot$, so by the IH (i), this case cannot arise.
- **Case** $(\vee E)$:
  $$\mathcal{D} = \dfrac{\Gamma \vdash e' : C_1 \vee C_2 \quad \Gamma, x{:}C_1 \vdash E[x] : A \quad \Gamma, y{:}C_2 \vdash E[y] : A}{\Gamma \vdash E[e'] : A}\ (\vee E)$$
  $E[e']$ value is given. By Lemma 2, $E[x]$ value and e' value. By the IH, either $\vdash [\sigma]e' : [\sigma]C_1$ or $\vdash [\sigma]e' : [\sigma]C_2$. Assume the first possibility, $\vdash [\sigma]e' : C_1$ (the second is symmetric). Let $\sigma' = \sigma, [\sigma]e'/x$; by (pvar-$\sigma$), $\vdash \sigma' : \Gamma, x{:}C_1$. By the IH, $\Gamma \vdash e' : C_1 \vee C_2$ is definite, so σ' is a definite substitution. $E[x]$ value so we can apply the IH to show that $[\sigma']\, E[x]$ has properties (i), (ii), (iii). But $[\sigma']\, E[x] = [\sigma]\, E[e']$, so $[\sigma]\, E[e']$ has properties (i), (ii), (iii).

$\square$

Lemma 5. *Every substitution σ such that $\vdash \sigma : \Gamma$ is definite.*

Proof. By induction on the derivation $\mathcal{D}$ of $\vdash \sigma : \Gamma$. The case for (empty-σ) is trivial. The cases for (prop-σ) and (ivar-σ) follow easily from the IH. For (pvar-σ) deriving $\vdash \sigma, v/x : \Gamma, x{:}A$, we have $\vdash v : [\sigma]A$ as a subderivation. Since v is closed, $v = [\sigma]v = [\sigma, v/x]x$, yielding $\vdash [\sigma, v/x]x : [\sigma]A$. The result follows by Lemma 4 applied with an empty substitution. □

Theorem 2 (Value definiteness). *If $\vdash \sigma : \Gamma$ and $\Gamma \vdash v : A$, then:*

(i) it is not the case that $\Gamma \vdash A \leq \bot$;
(ii) if $\Gamma \vdash A \leq A_1 \vee A_2$ then $\vdash [\sigma]v : [\sigma]A_1$ or $\vdash [\sigma]v : [\sigma]A_2$;
(iii) if $\Gamma \vdash A \leq \Sigma a{:}\gamma.\, A'$, there exists an index $\vdash i : \gamma$ where $\vdash [\sigma]\, v : [\sigma, i/a]\, A'$.

Proof. Follows immediately from Lemmas 4 and 5. □

For each ordinary type (not property types) we have a value inversion lemma (also known as *genericity* or *canonical forms*). We show only one example. Note the necessary generalization to allow for substitutions.

Lemma 6 (Inversion on $\rightarrow$).
If $\mathcal{D}$ derives $\Gamma \vdash v : B$ and $\Gamma \vdash B \leq B_1 \rightarrow B_2$ and $\vdash \sigma : \Gamma$ then $v = \lambda x.\, e$ where $\vdash [\sigma, v'/x]\, e : [\sigma]B_2$ for any $\vdash v' : [\sigma]B_1$.

Proof. By induction on $\mathcal{D}$. □

7 Type Preservation and Progress

Having proved value definiteness, we are ready to prove type safety. We prove the preservation and progress theorems simultaneously; we could prove them separately, but the proofs would share so much structure as to be more cumbersome than the simultaneous proof. (Our semantics is deterministic, so the combined form is meaningful.)

Theorem 3 (Type Preservation and Progress). *If $\Gamma \vdash e : C$ and σ is a substitution over program variables such that $\vdash \sigma : \Gamma$ and $\overline{\Gamma} = \cdot$, then either*

(1) e value and $\vdash [\sigma]\, e : C$, or
(2) there exists a term e' such that $[\sigma]\, e \mapsto e'$ and $\vdash e' : C$.

(By $\vdash \sigma : \Gamma$, σ substitutes no fixed point variables, so Γ must contain no fixed point variables. Moreover, $\overline{\Gamma} = \cdot$ so Γ contains no index variables.)

Proof. By induction on the derivation $\mathcal{D}$ of $\Gamma \vdash e : C$. Note that since Γ contains no index variables, $[\sigma]A = A$ for all types A. If e value, the result follows by Lemma 3. So suppose e is not a value. Rules ($\mathbf{1}I$), ($\rightarrow I$), ($\wedge I$), ($\top I$), (ΠI) and (var) can only type values. Γ types no fixed point variable, so (fixvar) cannot have been used. $\vdash \sigma : \Gamma$, so $\overline{\Gamma} \not\models \bot$: The cases for (sub) and (**fix**) use Lemma 3. The ($\rightarrow E$) case requires Lemmas 6 and 3. For ($\vee I_1$), ($\vee I_2$), (ΣI), ($\wedge E_1$), ($\wedge E_2$) simply apply the IH and reapply the rule. For (direct), ($\bot E$), ($\vee E$) and (ΣE), which type an evaluation context $E[e']$, we proceed thus:

– If the whole term $E[e']$ is a value, just apply Lemma 3.
– If e' is not a value:
  (1) apply the IH to $\Gamma \vdash e' : D$ to obtain $[\sigma]\, e' \mapsto e''$ with $\vdash e'' : D$;
  (2) from $[\sigma]\, e' \mapsto e''$, use (ev-context) to show $[\sigma]\, E[e'] \mapsto [\sigma]\, E[e'']$;
  (3) reapply the rule, with premise $\vdash e'' : D$, to yield $\vdash [\sigma]\, E[e''] : C$.
– If e' is a value (but $E[e']$ is not), use value definiteness (Theorem 2), yielding a contradiction for $(\bot E)$, or a new derivation for (direct), $(\vee E)$, (ΣE); in the latter cases apply the IH with substitution $[\sigma, [\sigma]e'/x]$.

The last subcase is the most interesting; we show it for $(\bot E)$ and $(\vee E)$. The (direct) and (ΣE) cases are similar.

– **Case $(\bot E)$**: $\mathcal{D} = \dfrac{\Gamma \vdash e' : \bot}{\Gamma \vdash E[e'] : C}\ (\bot E)$

  e' value and $\vdash \sigma : \Gamma$ are given. We have $\Gamma \vdash e' : \bot$ as a subderivation. By Theorem 2, $\Gamma \not\vdash e' : \bot$, a contradiction.
– **Case $(\vee E)$**: $\mathcal{D} = \dfrac{\Gamma \vdash e' : A \vee B \quad \Gamma, x{:}A \vdash E[x] : C \quad \Gamma, y{:}B \vdash E[y] : C}{\Gamma \vdash E[e'] : C}\ (\vee E)$

  e' value is given. We have $\Gamma \vdash e' : A \vee B$ as a subderivation. By Theorem 2, either $\vdash [\sigma]e' : A$ or $\vdash [\sigma]e' : B$. Since e' value, $[\sigma]e'$ value. Assume $\vdash [\sigma]e' : A$ (the other case is symmetric). $\Gamma, x{:}A \vdash E[x] : C$ is a subderivation. Let $\sigma' = \sigma, [\sigma]e'/x$. It is given that $\vdash \sigma : \Gamma$ and $\vdash [\sigma]\, e' : A$. By (pvar-σ), $\sigma' : \Gamma, x{:}A$, so by the IH, $[\sigma']\, E[x] \mapsto e''$ and $\vdash e'' : C$. But $[\sigma']\, E[x] = [\sigma, [\sigma']e'/x]\, E[x] = [\sigma]\, E[e']$, yielding $[\sigma]\, E[e'] \mapsto e''$.

$\square$

8 Related Work

The notion of datasort refinement combined with intersection types was introduced by Freeman and Pfenning [8]. They showed that full type inference was decidable under the so-called refinement restriction by using techniques from abstract interpretation. Interaction with effects in a call-by-value language was first addressed conclusively by Davies and Pfenning [6] which introduced the value restriction on intersection introduction, pointed out the unsoundness of distributivity, and proposed a practical bidirectional checking algorithm.

A different kind of refinement using indexed and dependent function types with indices drawn from a decidable constraint domain was proposed by Xi and Pfenning [17]. This language did not introduce pure property types, requiring syntactic markers for elimination of the existentials. Since this was unrealistic for many programs, Xi [15] presents an algorithm allowing existential elimination at every binding site after translation to a let-normal form. One can see some of the results in the current paper as a *post hoc* justification for this strategy (see the remarks at the end of Section 4.4).

Intersection types [4] were first incorporated into practical languages by Reynolds [12]. Pierce [11] gave examples of programming with intersection and union types in a pure λ-calculus using a type-checking mechanism that relied on syntactic markers. The first systematic study of unions in a type assignment framework by Barbanera, Dezani-Ciancaglini and de'Liguoro [2] identified a number of problems, including the failure of type preservation even for the pure λ-calculus when the union elimination rule is too unrestricted. It also provided a framework for our more specialized study of a call-by-value language with possible effects. van Bakel et al. [13] showed that the minimal relevant logic $\mathsf{B}+$ yields a type assignment system for the pure call-by-value λ-calculus; conjunction and disjunction become intersection and union, respectively. In their $\vee$-elimination rule, the subexpression may appear multiple times but must be a value; this rule is sound but impractical (see Section 4.1).

Some work on program analysis in compilation uses forms of intersection and union types to infer control flow properties [14, 10]. Because of the goals of these systems for program analysis and control flow information, the specific forms of intersection and union types are quite different from the ones considered here. Systems of *soft typing* designed for type inference in dynamically typed languages [3] are somewhat similar and also allow intersection, union, and even conditional types [1]. Again, however, the different setting and goals mean that the technical realization differs substantially from our proposal here.

9 Conclusion

We have designed a system of property types for the purpose of checking program invariants in call-by-value languages. We have presented the system as it was designed: incrementally, with each type constructor added orthogonally to an intermediate system that is itself sound and logically motivated. For both the definite and indefinite types, we have formulated rules that are not only sound but internally regular: the differences among $(\vee E)$, $(\perp E)$, (ΣE), (direct) are logical consequences of the type constructor's arity. The remarkable feature shared by all four rules is that *typing proceeds in evaluation order*, constituting a less *ad hoc* alternative to Xi's conversion to let-normal form. Lastly, we have formulated properties of *definiteness* of judgments, substitutions, and values, vital for our proof of type safety.

The pure type assignment system presented here is undecidable. We are in the process of developing a decidable bidirectional version (extending the system in [7], which did not include $\vee$ and Σ). The present system can be used to verify progress and preservation after erasure of all type annotations, and will be the basis of soundness in the bidirectional system. In particular, it verifies that type-checked programs do not need to carry types at runtime.

The major items of future work are the development of an efficient algorithm for type-checking and the evaluation of the pragmatics of the system in a full-scale language. While we have elided any explicit effects from the present system for the sake of brevity, the analysis in [6] applies to this setting and the present system. Moreover, since parametric polymorphism was orthogonal in the system of [6], we expect polymorphism will be orthogonal here as well. Ultimately, the system must be justified not only by its soundness and internal design but by its effectiveness in checking interesting properties of real programs.

Acknowledgments. This work is supported in part by the National Science Foundation under grants ITR/SY+SI 0121633 *Language Technology for Trustless Software Dissemination* and CCR-0204248 *Type Refinements*. In addition, the first author is supported in part by an NSF Graduate Research Fellowship. Brigitte Pientka and the anonymous referees provided insightful feedback.

References

1. Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proc. 21st Symposium on Principles of Programming Languages (POPL '94)*, pages 163–173, 1994.
2. Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo de'Liguoro. Intersection and union types: syntax and semantics. *Inf. and Comp.*, 119:202–230, 1995.
3. R. Cartwright and M. Fagan. Soft typing. In *Proc. SIGPLAN '91 Conf. Programming Language Design and Impl. (PLDI)*, volume 26, pages 278–292, 1991.
4. M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional characters of solvable terms. *Zeitschrift f. math. Logik und Grundlagen d. Math.*, 27:45–58, 1981.
5. Rowan Davies. Practical refinement-type checking. PhD thesis proposal, Carnegie Mellon University, 1997.
6. Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *Proc. Int'l Conf. Functional Programming (ICFP '00)*, pages 198–208, 2000.
7. Joshua Dunfield. Combining two forms of type refinements. Technical Report CMU-CS-02-182, Carnegie Mellon University, September 2002.
8. Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proc. SIGPLAN '91 Conf. Programming Language Design and Impl. (PLDI)*, volume 26, pages 268–277. ACM Press, June 1991.
9. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
10. Jens Palsberg and Christina Pavlopoulou. From polyvariant flow information to intersection and union types. *J. Functional Programming*, 11(3):263–317, 2001.
11. Benjamin C. Pierce. Programming with intersection types, union types, and polymorphism. Technical Report CMU-CS-91-106, Carnegie Mellon University, 1991.
12. John C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, 1996.
13. S. van Bakel, M. Dezani-Ciancaglini, U. de'Liguoro, and Y. Motohoma. The minimal relevant logic and the call-by-value lambda calculus. To appear, July 1999.
14. J.B. Wells, Allyn Dimock, Robert Muller, and Franklyn Turbak. A calculus with polymorphic and polyvariant flow types. *J. Functional Programming*, 12(3):183–317, May 2002.
15. Hongwei Xi. *Dependent types in practical programming*. PhD thesis, Carnegie Mellon University, 1998.
16. Hongwei Xi. Dependently typed data structures. Revised version superseding that of WAAAPL '99, available electronically, February 2000.
17. Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proc. 26th Symp. on Principles of Programming Languages (POPL'99)*, pages 214–227. ACM Press, 1999.