Imperative Programming with Dependent Types

(*Extended Abstract*)

Hongwei Xi
University of Cincinnati
hwxi@ececs.uc.edu

Abstract

In this paper, we enrich imperative programming with a form of dependent types. We start with explaining some motivations for this enrichment and mentioning some major obstacles that need to be overcome. We then present the design of a source level dependently typed imperative programming language Xanadu*, forming both static and dynamic semantics and then establishing the type soundness theorem. We also present realistic examples, which have all been verified in a prototype implementation, in support of the practicality of Xanadu. We claim that the language design of Xanadu is novel and it serves as an informative example that demonstrates a means to combine imperative programming with dependent types.*

1 Introduction

In [16, 8], the functional programming language ML is extended with a restricted form of dependent types. This extension yields a dependently typed functional programming language DML, in which the programmer can use dependent types to more accurately capture program properties and thus detect more program errors at compile-time. It is also demonstrated that dependent types in DML can facilitate array bound check elimination [15], redundant pattern matching clause removal [9], tag check elimination and untagged representation of datatypes [10]. Evidently, an immediate question is whether we can reap some similar benefits by introducing dependent types into imperative programming. We give a positive answer to this question in this paper.

Intuitively speaking, dependent types are types which depend on the values of language expressions. For instance, we can refine the type `int` for integers into infinitely many singleton types $\texttt{int}(a)$, where a, ranging over all integers, is the expression on which this type depends. Also, we can form a type $\texttt{int array}(a)$ for integer arrays of size a, where a ranges over all natural numbers. We cannot rely directly on standard systems of dependent types [4] for languages with computational effects. For instance, it is entirely unclear what it means to say A has type $\texttt{int array}(a)$ for some *mutable* variable a: if we update the value of a, this changes the type of A but A itself is unchanged. As in DML, we introduce a clear separation between ordinary run-time expressions and a distinguished family of index expressions, linked by singleton types of form $\texttt{int}(a)$: every expression of type $\texttt{int}(a)$ must evaluate to a (if the evaluation terminates). This decision is also crucial to obtaining a practical type-checking algorithm.

There is yet another obstacle. Suppose we have already declared that a mutable variable x has type $\texttt{int}(a)$ for some a. Usually, the type of a variable is fixed upon declaration in a programming language. This means that we cannot even type the assignment $x := x + 1$ since x and $x + 1$ have different types $\texttt{int}(a)$ and $\texttt{int}(a+1)$, respectively, making dependent types largely useless in imperative programming. In order to type $x := x + 1$, we must allow the type of x to change from $\texttt{int}(a)$ into $\texttt{int}(a + 1)$ after evaluating the assignment.

We propose to allow the type of a variable to change during evaluation and study some consequences of this proposal in the design of a source level programming language. Note that a similar approach is already used in Typed Assembly Language (TAL) [5], where the type of a register r may change during execution so as to reflect the type of the content of r at different program points.

We present some introductory examples on imperative programming with dependent types before going into further details, giving the reader some feel as to what dependent types can actually do in practice. These examples enable us to identify certain ways of using dependent types in practical programming, which we also regard as a major contribution of the paper.

In Figure 1, the Xanadu program implements a binary search function on an integer array. We use `int[0,n]` as a shorthand for `[i:int | 0<=i<=n] int(i)`. Note that the concrete syntax `[i:int | 0<=i<=n]` stands for a dependent sum constructor, usually written as $\Sigma i : \gamma$, where γ is the subset sort $\{a : int \mid 0 \leq a \leq n\}$, that is, the sort for all index expressions between 0 and n. In short, `int[0,n]` is the type for all integers i satisfying $0 \leq i \leq n$. Similarly, we use `int[-1,n)` for all integers

```
{n:nat}
int bsearch(key: int, vec[n]: int) {
  var:
    low: int[0,n]; high: int[-1,n);
    int mid, x;;

  low = 0; high = arraysize(vec) - 1;

  while (low <= high) {
    mid = (low + high) / 2;
    x = vec[mid];
    if (key == x) { return mid; }
    else if (key < x) { high = mid-1; }
         else { low = mid+1; }
  }
  return -1;
}
```

Figure 1. Binary search in Xanadu

j satisfying $-1 \leq j < n$.

The declared type `int[0,n]` for variable `low`, which will be called the master type of `low`, means that we can only store an integer i satisfying $0 \leq i \leq n$ into variable `low`, where n is the size of the array in `vec`. The declared type for variable `high` can be interpreted similarly. It can be proven in the type system of Xanadu that the array subscripting `vec[mid]` in the program is always safe at run-time, that is, the integer in `mid` is always within the bounds of the array in `vec` when the subscripting is performed. We will briefly explain the reason for this in Section 4. Therefore, it is unnecessary to insert run-time array bound checks when we compile the program.

In Figure 2, we first declare a polymorphic union type to represent lists. A union type in Xanadu is the same as a datatype in ML. In this case, the declaration indicates that we index the union type with an index of sort `nat`, which stands for the length of a list in this case. An index of sort `nat` basically means that the index represents a natural number.

- `Nil(0)` indicates that `Nil` is assigned the type `<'a>list(0)`, that is, `Nil` is a list of length 0.

- `{n:nat} Cons(n+1) of 'a * <'a>list(n)` means that `Cons` is given the following type

  ```
  {n:nat}
    'a * <'a>list(n) -> <'a>list(n+1)
  ```

  that is, `Cons` yields a list of length $n + 1$ when given a pair consisting of an element and a list of length n. We write `{n:nat}` for the dependent function type constructor, usually written as $\Pi n : nat$, which can also be seen as a universal quantifier.

```
union <'a>list of nat {
 Nil(0);
 {n:nat} Cons(n+1) of 'a * <'a>list(n);
}

('a){m:nat, n:nat}
<'a>list(m+n)
revApp(xs:<'a>list(m),ys:<'a>list(n)) {
   var: 'a x;;

   invariant:
     [m1:nat,n1:nat | m1+n1 = m+n]
     (xs:<'a>list(m1), ys:<'a>list(n1))
   while (true) {
     switch (xs) {
       case Nil: return ys;
       case Cons(x,xs): ys = Cons(x,ys);
     }
   }
   exit; /* can never be reached */
}

('a){n:nat}
<'a>list(n) reverse (xs: <'a>list(n)) {
   return revApp(xs, Nil);
}
```

Figure 2. The list reverse function in Xanadu

We then define the function `revApp`, which takes a pair of lists (xs, ys) and returns a list that is the concatenation of the reverse of xs to ys. The header of the function indicates that `revApp` yields a list of length $m + n$ when given a pair of lists of lengths m and n, respectively. The syntax following the keyword `invariant` denotes a *state type*, which can be regarded as a form of loop invariant. The state type means that the variables `xs` and `ys` must have types `<'a>list(m1)` and `<'a>list(n1)` at the program point immediately before the loop, where `m1` and `n1` are natural numbers satisfying `m1+n1 = m+n`. The `switch` statement inside the loop corresponds to a case statement in ML. For instance, suppose the clause:

```
case Cons(x, xs):  ys = Cons(x, ys)
```

is chosen at run-time; the head and tail of `xs` are then assigned to `x` and `xs`, respectively; `Cons(x, ys)` is assigned to `ys` and the loop repeats. The function `reverse` is defined as a special case of `revApp` and its header indicates that this function returns a list of length n when given one of length n.

We present another example in Figure 3 to demonstrate some use of dependently typed record. We define a polymorphic record `<'a>sparseArray(m,n)` for representing two-dimensional sparse arrays of dimension `m` by `n` in which each element is of type `'a`. Let `r` be a record

```
{m:nat,n:nat}
record <'a>sparseArray(m, n) {
  row: int(m); /* number of rows */
  col: int(n); /* number of columns */
  data[m]: <int[0,n) * 'a>list
  /*array of lists representing rows*/
}
```

Figure 3. A dependent record type

of type `<'a>sparseArray(m, n)`. Then `r` has three components, namely, `row`, `col` and `data`. Clearly, the types assigned to `row` and `col` indicate that `r.row` and `r.col` return the dimensions of `r`. The type assigned to `data` states that `r.data` is an array of size m. In this array, each element, which represents a row in a sparse array, is a list of pairs and each pair consists of a natural number less than n and an element of type `'a`. For instance, a list consisting of two pairs $(6, 2.7183)$ and $(23, 3.1416)$ represents a row in a sparse array where the 6th and 23rd elements are 2.7183 and 3.1416, respectively, and the rest are 0.0. A Xanadu program implementing the multiplication between a sparse array and a vector can be found at [13], in which all array subscripting is proven safe in the type system of Xanadu.

There is another important motivation behind the design of Xanadu. In an untrusted computing environment such as the Internet, a code recipient may not trust the origin of received mobile code. This makes array bound check elimination significantly more difficult since we need not only to eliminate array bound checks but also to convince the code recipient that the elimination is done correctly. The notion of proof-carrying code [6] can address the difficulty by attaching to mobile code a proof asserting that the code can never perform out-of-bounds array subscripting at run-time. The code recipient can then verify the attached proof independently and execute the code with no run-time array bound checking if the proof verification is successful.

However, there remains a challenging question with this approach: *how can we generate such a proof in the first place?* The Touchstone compiler [7], which compiles programs in a type-safe subset of C into proof-carrying code, handles this question with loop invariant synthesis in source programs. This is a fully automatic approach, but it is highly heuristic and can be too limited in practice. For instance, the Touchstone compiler seems unable to handle the binary search example presented in Figure 1. As for the example of sparse matrix multiplication, it does not even seem clear how a heuristic approach can actually work in this case.

We have designed a dependently typed assembly language DTAL in [14]. The type system of DTAL is capable of capturing memory safety of code at assembly level, including both type safety and safe array subscripting. After receiving DTAL code, a code recipient uses type-checking to verify whether the received code is memory safe and executes the code with no run-time array bound checking if the type-checking is successful. We plan to compile Xanadu programs into DTAL code, using annotations in Xanadu programs, which are in the form of dependent types, to generate dependent types in DTAL code. This can be regarded as an alternative design to that of Touchstone. Some preliminary results, including a prototype compiler, have been reported in [14]. In general, the type system of Xanadu allows the programmer to use dependent types to capture program invariants more accurately, and this, in return, leads to more effective detection of program errors and more thorough elimination of array bound checks.

The most significant contribution of the paper is the design of Xanadu, a source level imperative programming language supporting a form of dependent types. This design includes forming both static and dynamic semantics for Xanadu and imposing restrictions to make Xanadu practical. We also establish the type soundness of Xanadu, which constitutes the main technical contribution of the paper. We view the design of Xanadu as an example that illustrates an approach to enriching imperative programming with dependent types.

The rest of paper is organized as follows. We form a language Xanadu_0 in Section 2, which allows the type of a variable to change during evaluation but supports no dependent types. The introduction of Xanadu_0 is mainly for setting up the machinery to reason about the language $\text{Xanadu}_0^{\Pi,\Sigma}$, which essentially extends Xanadu_0 with a restricted form of dependent types. We present $\text{Xanadu}_0^{\Pi,\Sigma}$ in Section 3, where we also form both static and dynamic semantics for $\text{Xanadu}_0^{\Pi,\Sigma}$ and establish its type soundness. We then discuss in Section 4 some key issues on the design of an external language *Xanadu* and a type inference algorithm for it. Section 5 deals with some restrictions and extensions that make Xanadu more realistic. Lastly, we mention a prototype implementation, discuss some related work and conclude. We refer the interested reader to [12] for the details omitted here.

We focus on the technical development of Xanadu in this abstract, which is considerably involved. A short and intuitive introduction on Xanadu can be found in [11], where various programming examples are presented with explanation. Also, it could be helpful if the reader would briefly read Section 4 before studying Section 2 and Section 3 so as to get a feel as to how type inference is performed in Xanadu.

2 Xanadu_0

We start with a simple programming language Xanadu_0 and present its syntax in Figure 4. For instance, the following program in C:

```
int fact(int x) {
  return ((x > 0)? x * fact (x-1) : 1);
}
```

types	τ	$::=$	$\texttt{bool} \mid \texttt{int} \mid \tau\ \texttt{array} \mid \texttt{unit} \mid \texttt{top}$
function types	λ	$::=$	$(\tau_1, \ldots, \tau_n) \rightarrow \tau$
constants	c	$::=$	$b \mid n \mid \langle\rangle$
expressions	e	$::=$	$c \mid x \mid X \mid op(e_1, \ldots, e_n) \mid x := e \mid e_1; e_2 \mid \mathbf{if}(e, e_1, e_2) \mid \mathbf{while}(e_1, e_2) \mid$ $\mathbf{alloc}(e_1, e_2) \mid \mathbf{arraysize}\ (e) \mid e_1[e_2] \mid e[e_1] := e_2 \mid$ $\mathbf{let}\ X = e_1\ \mathbf{in}\ e_2\ \mathbf{end} \mid \mathbf{newvar}\ x\ \mathbf{in}\ e\ \mathbf{end} \mid \mathbf{call}(X; e_1, \ldots, e_n)$
values	v	$::=$	$X \mid c$
functions	f	$::=$	$\lambda(X_1, \ldots, X_n).e$
declarations	D	$::=$	$[] \mid D[X \mapsto f : \lambda]$
value variable contexts	Γ	$::=$	$\cdot \mid \Gamma, X : \lambda \mid \Gamma, X : \tau$
reference variable contexts	Δ	$::=$	$\cdot \mid \Delta, x : \tau$
programs	P	$::=$	$\mathbf{letdef}\ D\ \mathbf{in}\ e\ \mathbf{end}$

Figure 4. The syntax for Xanadu_0

is equivalent to extending a declaration D in Xanadu_0 with the following binding:

$$[X \mapsto \lambda(X_1).body : (\texttt{int}) \rightarrow \texttt{int}],$$

where *body* is

$$\begin{array}{l} \mathbf{newvar}\ x\ \mathbf{in} \\ \quad x := X_1; \mathbf{if}(x > 0, x * \mathbf{call}(X; x-1), 1) \\ \mathbf{end} \end{array}$$

There are two kinds of variables in Xanadu_0. We use x for reference variables and X for value variables. Basically, a reference variable is like a variable in imperative programming while a value variable is like one in (call-by-value) functional programming. We find that value variables are convenient for certain theoretical purposes but they are *not* indispensable. [1]

We use *op* for some primitive operations such as arithmetic and boolean operations. For a reference variable context Δ, the domain $\mathbf{dom}(\Delta)$ is the set of reference variables declared in Δ. The domain $\mathbf{dom}(\Gamma)$ of a value variable context Γ is defined similarly. We require that no reference (value) variables appear more than once in Δ (Γ). Also, we use $\mathbf{call}(X; e_1, \ldots, e_n)$ to indicate a function call in Xanadu_0, where X is assumed to be bound to a function taking n arguments.

The most significant feature of Xanadu_0 is its type system, in which the type of a reference variable is allowed to change during evaluation. We emphasize at this point that Xanadu_0 is not intended for demonstrating the advantage of a language that allows the type of a reference variable to change during evaluation. Such advantage seems unclear (if there is any) until the introduction of dependent types. The design of Xanadu_0 is primarily for setting up some machinery needed to reason about $\text{Xanadu}_0^{\Pi,\Sigma}$, which is to be introduced in Section 3, allowing for a less involved presentation.

[1] Another reason for having value variables is that we may also support various functional programming features in Xanadu in future.

2.1 Static Semantics

Xanadu_0 is a monomorphically typed first-order language. The only subtyping rules in Xanadu_0 are the following.

$$\frac{}{\models \tau \leq \tau}\ \textbf{(co-eq)} \qquad \frac{}{\models \tau \leq \texttt{top}}\ \textbf{(co-top)}$$

Given two reference variable contexts Δ_0 and Δ_1, we have the following rule **(co-context)** for coercing Δ_0 into Δ_1, where we assume $\mathbf{dom}(\Delta_0) = \mathbf{dom}(\Delta_1)$.

$$\frac{\models \Delta_0(x) \leq \Delta_1(x) \quad \text{for all } x \in \mathbf{dom}(\Delta_0)}{\Delta_0 \models \Delta_1}$$

A typing judgment in Xanadu_0 is of form $\Delta_0; \Gamma \vdash e : (\Delta_1; \tau)$, which means that the expression e has type τ under the context $\Delta_0; \Gamma$ and the evaluation of e changes Δ_0 into Δ_1. For instance, we will see that the following is derivable,

$$x : \texttt{int}; \cdot \vdash (x := \mathbf{true}) : (x : \texttt{bool}; \texttt{unit}),$$

which clearly indicates that the type of x changes from $\texttt{int}$ into $\texttt{bool}$ after the assignment $x := \mathbf{true}$.

We present some of the typing rules for Xanadu_0 in Figure 5. Given a function $f = \lambda(X_1, \ldots, X_n).e$ and a function type $\lambda = (\tau_1, \ldots, \tau_n) \rightarrow \tau$, the rule **(type-function)** is for typing a function. We emphasize that only toplevel functions are allowed in Xanadu_0. Notice that the rule **(type-newvar)** assigns every uninitialized variable the type $\texttt{top}$. This implies that a variable is already initialized if it has a type other than $\texttt{top}$.

In Xanadu_0, the type of a reference variable can change during evaluation. For instance, the following function can be given the type $(\texttt{int}, \texttt{int}) \rightarrow \texttt{bool}$ in Xanadu_0. Note that the type of x changes from $\texttt{top}$ into $\texttt{int}$ and then into

$$\frac{\Delta_0; \Gamma \vdash e : (\Delta_1; \tau)}{\Delta_0; \Gamma \vdash x := e : (\Delta_1[x \mapsto \tau]; \texttt{unit})} \textbf{ (type-assign)}$$

$$\frac{\Delta_0; \Gamma \vdash e : (\Delta_1; \texttt{bool}) \quad \Delta_1; \Gamma \vdash e_1 : (\Delta_2; \texttt{unit}) \quad \Delta_2 \models \Delta \quad \Delta_1; \Gamma \vdash e_2 : (\Delta_3; \texttt{unit}) \quad \Delta_3 \models \Delta}{\Delta_0; \Gamma \vdash \mathbf{if}(e, e_1, e_2) : (\Delta; \texttt{unit})} \textbf{ (type-if)}$$

$$\frac{\Delta_0 \models \Delta \quad \Delta; \Gamma \vdash e_1 : (\Delta_1; \texttt{bool}) \quad \Delta_1; \Gamma \vdash e_2 : (\Delta_2; \texttt{unit}) \quad \Delta_2 \models \Delta}{\Delta_0; \Gamma \vdash \mathbf{while}(e_1, e_2) : (\Delta_1; \texttt{unit})} \textbf{ (type-while)}$$

$$\frac{\Delta_0; \Gamma \vdash e_1 : (\Delta_1; \texttt{int}) \quad \Delta_1; \Gamma \vdash e_2 : (\Delta_2; \tau)}{\Delta_0; \Gamma \vdash \mathbf{alloc}(e_1, e_2) : (\Delta_2; \tau \texttt{ array})} \textbf{ (type-alloc)}$$

$$\frac{\Gamma(X) = (\tau_1, \ldots, \tau_n) \to \tau \quad \Delta_0; \Gamma \vdash e_1 : (\Delta_1; \tau_1) \quad \cdots \quad \Delta_{n-1}; \Gamma \vdash e_n : (\Delta_n; \tau_n)}{\Delta_0; \Gamma \vdash \mathbf{call}(X; e_1, \ldots, e_n) : (\Delta_n; \tau)} \textbf{ (type-call)}$$

$$\frac{\Delta_0, x : \texttt{top}; \Gamma \vdash e : (\Delta_1, x : \tau_1; \tau_2)}{\Delta_0; \Gamma \vdash \mathbf{newvar}\ x\ \mathbf{in}\ e\ \mathbf{end} : (\Delta_1; \tau_2)} \textbf{ (type-newvar)}$$

$$\frac{\cdot; \Gamma, X_1 : \tau_1, \ldots, X_n : \tau_n \vdash e : (\cdot; \tau)}{\Gamma \vdash \lambda(X_1, \ldots, X_n).e : (\tau_1, \ldots, \tau_n) \to \tau} \textbf{ (type-function)}$$

Figure 5. Some typing rules for Xanadu_0

`bool` during evaluation.

$$\begin{array}{l} \lambda(X_1, X_2). \\ \quad \mathbf{newvar}\ x\ \mathbf{in} \\ \quad\quad x := X_1 - X_2; \mathbf{if}(x > 0, x := \mathbf{true}, x := \mathbf{false}); x \\ \quad \mathbf{end} \end{array}$$

In the rule **(type-assign)**, we use $\Delta_1[x \mapsto \tau]$ for a finite map Δ such that $\Delta(x) = \tau$ and $\Delta(y) = \Delta_1(y)$ for all other variables y in $\mathbf{dom}(\Delta) = \mathbf{dom}(\Delta_1)$. The rule **(type-while)** needs some explanation. The reference variable context Δ in the rule can essentially be regarded as a loop invariant on the types of reference variables in the loop that must hold at the beginning of the loop. In Xanadu_0, the programmer is responsible for providing such loop invariants. One may argue that this practice is too burdensome for the programmer. However, we feel that this argument is less tenable because (a) it already suffices to provide invariants only for those variables whose types may change in a loop and (b) it seems most likely that there are only few such variables. In particular, there is no need for such invariants if there are no variables whose types change during evaluation. The programmer, who is allowed to change the types of variables when programming in Xanadu_0, can always choose not to do so and thus provide no loop invariants.

A real serious problem with Xanadu_0 is in compilation. If we can assign to a reference variable a value of any type, we need a uniform representation for values of all types, that is, we need to box all values which cannot be represented in a word on a real machine. This is similar to supporting polymorphism in a language. Given that the advantage of Xanadu_0 is unclear, we feel that the cost of boxed representation is simply too high. However, the introduction of a restricted form of dependent types into Xanadu_0 will completely alter the situation as illustrated in Section 3.

2.2 Dynamic Semantics

We form an abstract machine for assigning dynamic semantics to Xanadu_0. A machine state $\mathcal{M}$ is a pair of finite mappings $\langle \mathcal{V}, \mathcal{H} \rangle$. The domain $\mathbf{dom}(\mathcal{H})$ of $\mathcal{H}$ is a set of heap addresses, which one may assume are represented as natural numbers. For every $h \in \mathbf{dom}(\mathcal{H})$, $\mathcal{H}(h)$ is a tuple $(hc_0, \ldots, hc_{n-1})$, where we use hc for either a constant or a heap address. The domain $\mathbf{dom}(\mathcal{V})$ of $\mathcal{V}$ is a set of reference variables and $\mathcal{V}$ maps x to some hc for every $x \in \mathbf{dom}(V)$. A judgment of form $\mathcal{M}[e] \to_D \mathcal{M}'[e']$ means that expression e under machine state $\mathcal{M}$ evaluates to e' under $\mathcal{M}'$, where declaration D binds the function symbols in e to some functions. Notice that heap addresses can creep into expressions during evaluation. [2] We thus extend the syntax for Xanadu_0 to treat heap addresses as constants. This enables us to form expressions involving heap addresses, which are needed for forming evaluation rules.

We also introduce expressions of the following form

$$\mathbf{newvar}\ x = hc\ \mathbf{in}\ e\ \mathbf{end}$$

and treat $\mathbf{newvar}\ x\ \mathbf{in}\ e\ \mathbf{end}$ as

$$\mathbf{newvar}\ x = \langle\rangle\ \mathbf{in}\ e\ \mathbf{end}.$$

The typing rule **(type-newvar)** for

$$\mathbf{newvar}\ x = hc\ \mathbf{in}\ e\ \mathbf{end}$$

[2] This happens when the rule **(eval-alloc)** is applied.

$$\frac{}{\langle \mathcal{V}, \mathcal{H} \rangle [\mathbf{newvar}\ x = hc_1\ \mathbf{in}\ hc_2\ \mathbf{end}] \rightarrow_D \langle \mathcal{V}, \mathcal{H} \rangle [hc_2]}\ \textbf{(eval-newvar-2)}$$

$$\frac{}{\mathcal{M}[\mathbf{while}(e_1, e_2)] \rightarrow_D \mathcal{M}[\mathbf{if}(e_1, (e_2; \mathbf{while}(e_1, e_2)), \langle\rangle)]}\ \textbf{(eval-while)}$$

$$\frac{n \geq 0 \quad h \notin \mathbf{dom}(\mathcal{H})}{\langle V, \mathcal{H} \rangle [\mathbf{alloc}(n, hc)] \rightarrow_D \langle \mathcal{V}, \mathcal{H}[h \mapsto (hc, \ldots, hc)] \rangle [h]}\ \textbf{(eval-alloc-3)}$$

$$\frac{\mathcal{H}(h) = (hc_0, \ldots, hc_{k-1}) \quad 0 \leq n < k}{\langle \mathcal{V}, \mathcal{H} \rangle [h[n]] \rightarrow_D \langle \mathcal{V}, \mathcal{H} \rangle [hc_n]}\ \textbf{(eval-array-access-in)}$$

$$\frac{\mathcal{H}(h) = (hc_0, \ldots, hc_{k-1}) \quad n < 0 \vee k \leq n}{\langle \mathcal{V}, \mathcal{H} \rangle [h[n]] \rightarrow_D \mathbf{subscript}}\ \textbf{(eval-array-access-out)}$$

$$\frac{\begin{array}{c}\mathcal{H}(h) = (hc_0, \ldots, hc_{k-1}) \qquad 0 \leq n < k \\ \mathcal{H}' = \mathcal{H}[h \mapsto (hc_0, \ldots, hc_{n-1}, hc, hc_{n+1}, \ldots, hc_{k-1})]\end{array}}{\langle \mathcal{V}, \mathcal{H} \rangle [h[n] = hc] \rightarrow_D \langle \mathcal{V}, \mathcal{H}' \rangle [\langle\rangle]}\ \textbf{(eval-array-assign-in)}$$

$$\frac{\mathcal{H}(h) = (hc_0, \ldots, hc_{k-1}) \qquad n < 0 \vee k \leq n}{\langle \mathcal{V}, \mathcal{H} \rangle [h[n] = hc] \rightarrow_D \mathbf{subscript}}\ \textbf{(eval-array-assign-out)}$$

Figure 6. Some evaluation rules for Xanadu_0

is given below, where H is a finite mapping that maps heap addresses in e to types. More details on this is to be presented in Subsection 2.3.

$$\frac{\Delta_0, x : H(hc); \Gamma \vdash_H e : (\Delta_1, x : \tau_1; \tau_2)}{\Delta_0; \Gamma \vdash_H \mathbf{newvar}\ x = hc\ \mathbf{in}\ e\ \mathbf{end} : (\Delta_1; \tau_2)}$$

We list some of the evaluation rules for Xanadu_0 in Figure 6. We use $h \notin \mathbf{dom}(\mathcal{H})$ in the rule **(eval-alloc-3)** to mean that h is a new heap address. Note that an array is always initialized upon allocation. Also we use $\mathbf{subscript}$ to indicate that an out-of-bounds array subscripting exception has occurred during evaluation. The evaluation rules for propagating exception $\mathbf{subscript}$ are all omitted.

We use heap addresses to form expressions when assigning dynamic semantics to Xanadu_0. However, it is conceivable that there are other approaches to assigning dynamic semantics to Xanadu_0 that require no use of expressions containing heap addresses. Therefore, we do not include heap addresses as a part of Xanadu_0.

2.3 Type Soundness

When establishing the soundness for Xanadu_0, we need to type expressions containing heap addresses. For this purpose, we use H for a finite mapping from heap addresses to array types and change the form of a typing judgment into $\Delta_0; \Gamma \vdash_H e : (\Delta_1; \tau)$, where we assume that all heap addresses in e are in the domain $\mathbf{dom}(H)$ of H. This affects all the typing rules in Figure 5. Also we use the following rule for typing a heap address.

$$\frac{}{\Delta; \Gamma \vdash_H h : (\Delta; H(h))}\ \textbf{(type-heap-address)}$$

We use judgments $\mathcal{M} \models \Delta$ and $\mathcal{H} \models H$ to mean $\mathcal{M}$ models Δ and $\mathcal{H}$ models H, respectively. The precise meaning of these judgments follows from the rules in Figure 7. Also, we write $\mathcal{M} \models H$ if $\mathcal{M} = \langle \mathcal{V}, \mathcal{H} \rangle$ and $\mathcal{H} \models H$ holds.

Assume that $\Delta_0; \Gamma \vdash_H, e : (\Delta_1; \tau)$ is derivable. Also assume that $\mathcal{M}$ models Δ and H, that is, $\mathcal{M} \models \Delta$ and $\mathcal{M} \models H$ are derivable. In order to establish the type soundness for Xanadu_0, we need to prove that if $\mathcal{M}[e] \rightarrow_D \mathcal{M}'[e']$ then $\mathcal{M}' \models \Delta'$ and $\mathcal{M}' \models H'$ are derivable for some Δ' and $\mathcal{H}'$ such that $\Delta'; \Gamma \vdash e' : (\Delta_1; \tau)$ is derivable.

Unfortunately, it is impossible to establish this. For instance, let $\mathcal{M} = (\mathcal{V}, \mathcal{H})$ such that $\mathcal{V}(x_1) = \mathcal{V}(x_2) = h$ and $\mathcal{H}(h) = (0)$. Note that we use (0) for a tuple consisting of exactly one element 0. Clearly, $\mathcal{M} \models \Delta$ is derivable for $\Delta = x_1 : \texttt{int array}(1), x_2 : \texttt{top array}(1)$. Also note that $\Delta; \Gamma \vdash x_2[0] := \mathbf{true} : (\Delta; \texttt{unit})$ is derivable. Suppose that we evaluate $x_2[0] := \mathbf{true}$ under $\mathcal{M}$. This evaluation leads $\mathcal{M}$ into $\mathcal{M}' = (\mathcal{V}, \mathcal{H}')$, where $\mathcal{H}'(h) = (\mathbf{true})$, but $\mathcal{M}' \models \Delta$ is no longer derivable since $\mathcal{V}(x_1) = h$ and $\mathcal{H}'(h)$, which is $(\mathbf{true})$, is not an integer tuple.

We introduce the following notion of regularity to address the problem.

Definition 2.1 *(Regularity) Let $\mathcal{D}$ be a derivation of $\mathcal{M} \models$*

$$\frac{}{\mathcal{H} \models \langle\rangle : \texttt{unit}}\ \textbf{(model-unit)} \quad \frac{}{\mathcal{H} \models hc : \texttt{top}}\ \textbf{(model-top)} \quad \frac{}{\mathcal{H} \models b : \texttt{bool}}\ \textbf{(model-bool)} \quad \frac{}{\mathcal{H} \models i : \texttt{int}}\ \textbf{(model-int)}$$

$$\frac{\mathcal{H} \models \mathcal{V}(x) : \tau}{\langle \mathcal{V}, \mathcal{H} \rangle \models x : \tau}\ \textbf{(model-var)} \quad \frac{\mathcal{H}(h) = (hc_0, \ldots, hc_{n-1}) \quad \mathcal{H} \models hc_0 : \tau \quad \cdots \quad \mathcal{H} \models hc_{n-1} : \tau}{\mathcal{H} \models h : \tau\ \texttt{array}(n)}\ \textbf{(model-array)}$$

$$\frac{\mathcal{M} \models x : \Delta(x) \quad \text{for all } x \in \textbf{dom}(\Delta)}{\mathcal{M} \models \Delta}\ \textbf{(model-context)} \quad \frac{\mathcal{H} \models h : H(h) \quad \text{for all } h \in \textbf{dom}(H)}{\mathcal{H} \models H}\ \textbf{(model-heap)}$$

Figure 7. Modeling Rules for Xanadu_0

Δ. *If the following rule is applied in $\mathcal{D}$,*

$$\frac{\begin{array}{c}\mathcal{H}(h) = (hc_0, \ldots, hc_{n-1}) \\ \mathcal{H} \models hc_0 : \tau \quad \cdots \quad \mathcal{H} \models hc_{n-1} : \tau\end{array}}{\mathcal{H} \models h : \tau\ \texttt{array}(n)}\ \textbf{(model-array)}$$

we say $\mathcal{D}$ associates h with τ. $\mathcal{D}$ is a regular derivation if $\mathcal{D}$ associates each heap address h with at most one type.

We now argue that the above $\mathcal{M} = \langle \mathcal{V}, \mathcal{H} \rangle \models \Delta$ cannot have a regular derivations. Note that we must have a derivation of the following form in order to derive $\mathcal{M} \models x_1 : \texttt{int array}(1)$.

$$\frac{\mathcal{H}(h) = (1) \quad \mathcal{H} \models 1 : \texttt{int}}{\mathcal{H} \models h : \texttt{int array}(1)}\ \textbf{(model-array)}$$

Similarly, we also must have a derivation of the following form in order to derive $\mathcal{M} \models x_2 : \texttt{top array}(1)$.

$$\frac{\mathcal{H}(h) = (1) \quad \mathcal{H} \models 1 : \texttt{top}}{\mathcal{H} \models h : \texttt{top array}(1)}\ \textbf{(model-array)}$$

Therefore, $\mathcal{M} \models \Delta$ cannot have a regular derivation since any derivation of $\mathcal{M} \models \Delta$ must associate h with both `int` and `top`.

Lemma 2.2 *Assume that $\mathcal{M} \models \Delta_0$ has a regular derivation and $\mathcal{M} \models H$, $\Delta_0; \Gamma \vdash_H e : (\Delta_1; \tau)$ and $\mathcal{M}[e] \rightarrow_D \mathcal{M}'[e']$ are derivable, Then there exist Δ'_0 and H' extending H such that $\mathcal{M}' \models \Delta'_0$ has a regular derivation and both $\mathcal{M}' \models H'$ and $\Delta'_0; \Gamma \vdash_{H'} e' : (\Delta_1; \tau)$ are derivable.*

Proof This follows from a structural induction on the derivation $\mathcal{D}$ of $\Delta_0; \Gamma \vdash_H, e : (\Delta_1; \tau)$. ■

Theorem 2.3 *Let $P = \textbf{letdef}\ D\ \textbf{in}\ e\ \textbf{end}$ be a program such that $\vdash P$ is derivable; if $\mathcal{M}[e] \rightarrow^*_D \mathcal{M}'[e']$, then either e' is $\langle\rangle$, or $\mathcal{M}'[e'] \rightarrow_D$ **subscript**, or $\mathcal{M}'[e'] \rightarrow_D \mathcal{M}''[e'']$ for some $\mathcal{M}''$ and e''. In other words, the evaluation of a well-typed program in* Xanadu_0 *either terminates normally, or raises a subscript exception, or runs forever.*

Proof Obviously, $\mathcal{M} \models \cdot$ has a regular derivation. This theorem then follows from Lemma 2.2. ■

We are now ready to incorporate dependent types into imperative programming.

index objects	i, j	$::=$	$a \mid n \mid i + j \mid i - j \mid$
			$i * j \mid i \div j \mid$
index propositions	p	$::=$	$i < j \mid i \leq j \mid i = j \mid$
			$i \neq j \mid i \geq j \mid i > j \mid$
			$p_1 \wedge p_2 \mid p_1 \vee p_2$
index sorts	γ	$::=$	$int \mid \{a : \gamma \mid p\}$
index contexts	ϕ	$::=$	$\cdot \mid \phi, a : \gamma \mid \phi, p$
constraint	Φ	$::=$	$p \mid p \supset \Phi \mid$
			$\Phi_1 \wedge \Phi_2 \mid \forall a : \gamma . \Phi$

Figure 8. Syntax for some integer constraint domain

types	τ	$::=$	$\texttt{bool}(i) \mid \texttt{int}(i) \mid \texttt{unit} \mid$
			$\tau\ \texttt{array}(i) \mid \Sigma a : \gamma . \tau$
function types	λ	$::=$	$\cdots \mid \Pi\phi.(\tau_1, \ldots, \tau_n) \rightarrow \tau$
expressions	e	$::=$	
			$\cdots \mid \langle i \mid e \rangle \mid \textbf{unpack}\ e_1\ \textbf{as}\ \langle a \mid X \rangle\ \textbf{in}\ e_2\ \textbf{end}$
values	v	$::=$	$\cdots \mid \langle i \mid v \rangle$
state types	σ	$::=$	$\exists \phi . \Delta$

Figure 9. Syntax for $\mathrm{Xanadu}_0^{\Pi,\Sigma}$

3 $\mathrm{Xanadu}_0^{\Pi,\Sigma}$

In this section, we extend Xanadu_0 into a dependently typed programming language $\mathrm{Xanadu}_0^{\Pi,\Sigma}$, where the dependent types are of a restricted form as in DML [16, 8].

We fix an integer constraint domain in Figure 8 and restrict the type index expressions in $\mathrm{Xanadu}_0^{\Pi,\Sigma}$, namely, the expressions that can be used to index a type, to this domain. This is a sorted domain and subset sorts can be formed. For instance, we use *nat* for the subset sort $\{a : int \mid a \geq 0\}$. We emphasize that the sort int should not be confused with the type `int`. We say that a satisfaction relation $\phi \models \Phi$ is satisfiable if $(\phi)\Phi$ holds in the integer constraint domain, where $(\phi)\Phi$ is defined as follows.

$$(\cdot)\Phi = \Phi \qquad (\phi, a : int)\Phi = (\phi)\forall a : int.\Phi$$
$$(\phi, \{a : \gamma \mid p\})\Phi = (\phi, a : \gamma)(p \supset \Phi)$$
$$(\phi, p)\Phi = (\phi)(p \supset \Phi)$$

The additional syntax of $\text{Xanadu}_0^{\Pi,\Sigma}$ to that of Xanadu_0 is given in Figure 9. For every integer i, $\texttt{int}(i)$ is a singleton type such that the value of every expression of this type equals i. Similarly, $\texttt{bool}(1)$ and $\texttt{bool}(0)$ are singleton types for expressions with values equal to true and false, respectively. Also we use $\Sigma a : \gamma.\tau$ for a sum type. We omit the rules for forming legal types, which are standard. For instance, it is required that $\phi \models 0 \leq i \leq 1$ be satisfiable in order to form $\texttt{bool}(i)$, where we assume that all index variables in i are declared in ϕ.

Also, we define the erasure of a type τ as follows.

$$\|\texttt{unit}\| = \texttt{unit} \qquad \|\texttt{top}\| = \texttt{top}$$
$$\|\texttt{bool}(i)\| = \texttt{bool} \qquad \|\texttt{int}(i)\| = \texttt{int}$$
$$\|\tau\ \texttt{array}(i)\| = \|\tau\|\ \texttt{array} \qquad \|\Sigma a : \gamma.\tau\| = \|\tau\|$$

Note that $\texttt{int}$ is interpreted as $\Sigma a : int.\texttt{int}(a)$ and $\tau\ \texttt{array}$ as $\Sigma a : nat.\tau\ \texttt{array}(a)$.

The significance of type erasure is that for every well-typed program P in $\text{Xanadu}_0^{\Pi,\Sigma}$, if we replace each type in P with its erasure then P becomes a well-type program in Xanadu_0. By this, we say that $\text{Xanadu}_0^{\Pi,\Sigma}$ is a conservative extension of Xanadu_0. A program that is typable in $\text{Xanadu}_0^{\Pi,\Sigma}$ is already typable in Xanadu_0, but dependent types in $\text{Xanadu}_0^{\Pi,\Sigma}$ can allow the programmer to capture more program properties and thus lead to the construction of more robust programs (as more program errors are detected statically).

We use a judgment of form $\phi \vdash \tau : *$ to indicate that τ is a well-formed type under index variable context ϕ. The rules for deriving such judgments are standard and thus omitted. We use judgments $\phi \vdash \Delta[\text{ref ctx}]$ and $\phi \vdash \Gamma[\text{val ctx}]$ to mean that Δ and Γ are well-formed reference and value variable contexts, respectively. The rules for these judgments are also standard and thus omitted.

We omit the details on how substitution is performed, which is standard. Given a term $\bullet$ such as a type or a context, we use $\bullet[\theta]$ for the result from applying θ to $\bullet$. We introduce a judgment of form $\phi \vdash \theta : \phi'$ and present the following rules for deriving such judgments.

$$\frac{}{\phi \vdash [] : \cdot}\ \textbf{(sub-emp)}$$

$$\frac{\phi \vdash \theta : \phi' \quad \phi \vdash i : \gamma}{\phi \vdash \theta[a \mapsto i] : \phi', a : \gamma}\ \textbf{(sub-var)}$$

$$\frac{\phi \vdash \theta : \phi' \quad \phi \models p[\theta]}{\phi \vdash \theta : \phi', p}\ \textbf{(sub-prop)}$$

Roughly speaking, $\phi \vdash \theta : \phi'$ means that θ has "type" ϕ' under ϕ.

3.1 Static Semantics

A typing judgment in $\text{Xanadu}_0^{\Pi,\Sigma}$ is of the following form, which is considerably involved and thus deserves some detailed explanation.

$$\phi_1; \Delta_1; \Gamma \vdash e : (\phi_2; \Delta_2; \tau)$$

There are several invariants associated with such as a typing judgment.

1. All reference variables in e are declared in Δ_1.
2. All value variables in e are declared in Γ.
3. $\phi_1 \vdash \Delta_1[\text{ref ctx}]$ is derivable.
4. $\phi_1 \vdash \Gamma[\text{val ctx}]$ is derivable.
5. ϕ_2 is an extension of ϕ_1, i.e., $\phi_2 = \phi_1, \phi$ for some ϕ.
6. $\phi_2 \vdash \Delta_2[\text{ref ctx}]$ is derivable.
7. $\phi_2 \vdash \tau : *$ is derivable.

Essentially, $\phi_1; \Delta_1; \Gamma \vdash e : (\phi_2; \Delta_2; \tau)$ means that for each substitution θ_1 satisfying $\cdot \vdash \theta_1 : \phi_1$ there exists a substitution θ_2 satisfying $\cdot \vdash \theta_2 : \phi_2$ such that $e[\theta_1]$ can be given type $\tau[\theta_2]$ under $\Delta_1[\theta_1]; \Gamma[\theta_1]$ and the evaluation of $e[\theta_1]$ changes $\Delta_1[\theta_1]$ into $\Delta_2[\theta_2]$.

Some of the typing rules for $\text{Xanadu}_0^{\Pi,\Sigma}$ are presented in Figure 10. In both rules **(type-if)** and **(type-while)**, the state type $\exists\phi.\Delta$ is to be either provided or synthesized.

3.2 Dynamic Semantics

The dynamic semantics of $\text{Xanadu}_0^{\Pi,\Sigma}$ is formed on top of that of Xanadu_0. We no longer need rules like **(eval-array-access-out)** and **(eval-array-assign-out)** since the type system of $\text{Xanadu}_0^{\Pi,\Sigma}$ is designed to guarantee that the evaluation of a well-typed program in $\text{Xanadu}_0^{\Pi,\Sigma}$ can never lead to out-of-bounds array subscripting.[3] We need some additional rules in Figure 11 for handling new language constructs.

3.3 Type Equality and Coercion

A judgment of form $\phi; \Delta \models \exists\phi'.\Delta'$ basically means that Δ coerces into $\Delta'[\theta]$, that is, Δ' under the substitution θ, for some θ satisfying $\phi \vdash \theta : \phi'$. We present rules for deriving such judgments in this section. In the presence of dependent types, it is no longer trivial to determine whether two types are equivalent. For instance, we have to prove the constraint $1 + 1 = 2$ in order to claim $\texttt{int}(1 + 1)$ is equivalent to

[3]Instead, the programmer is required to insert dynamic array bound checks in case an array index cannot be proven within the bounds of the indexed array.

$$\frac{\phi_1; \Delta_1; \Gamma \vdash e : (\phi_2; \Delta_2; \tau_1) \quad \phi_2 \models \tau_1 = \tau_2}{\phi_1; \Delta_1; \Gamma \vdash e : (\phi_2; \Delta_2; \tau_2)} \textbf{ (type-eq)}$$

$$\frac{\Delta(x) = \tau}{\phi; \Delta; \Gamma \vdash x : (\phi; \Delta; \tau)} \textbf{ (type-var)} \qquad \frac{\Delta(x) = \Sigma a : \gamma.\tau}{\phi; \Delta; \Gamma \vdash x : (\phi, a : \gamma; \Delta; \tau)} \textbf{ (type-open)}$$

$$\frac{\phi_1; \Delta_1; \Gamma \vdash e : (\phi_2; \Delta_2; \tau)}{\phi_1; \Delta_1; \Gamma \vdash x := e : (\phi_2; \Delta_2[x \mapsto \tau]; \texttt{unit})} \textbf{ (type-assign)}$$

$$\frac{\begin{array}{c}\Gamma(X) = \Pi\phi.(\tau_1, \ldots, \tau_n) \to \tau \quad \phi_1 \vdash \theta : \phi \\ \phi_1; \Delta_1; \Gamma \vdash e_1 : (\phi_2; \Delta_2; \tau_1[\theta]) \\ \cdots \\ \phi_n; \Delta_n; \Gamma \vdash e_n : (\phi_{n+1}; \Delta_{n+1}; \tau_n[\theta])\end{array}}{\phi_1; \Delta_1; \Gamma \vdash \mathbf{call}(X; e_1, \ldots, e_n) : (\phi_{n+1}; \Delta_{n+1}; \tau[\theta])} \textbf{ (type-call)}$$

$$\frac{\begin{array}{c}\phi_1; \Delta_1; \Gamma \vdash e : (\phi_2; \Delta_2; \texttt{bool}(i)) \\ \phi_2, i = 1; \Delta_2; \Gamma \vdash e_1 : (\phi_3; \Delta_3; \tau) \quad \phi_3; \Delta_3 \models \exists\phi.\Delta \\ \phi_2, i = 0; \Delta_2; \Gamma \vdash e_2 : (\phi_4; \Delta_4; \tau) \quad \phi_4; \Delta_4 \models \exists\phi.\Delta\end{array}}{\phi_1; \Delta_1; \Gamma \vdash \mathbf{if}(e, e_1, e_2) : (\phi_2, \phi; \Delta; \tau)} \textbf{ (type-if)}$$

$$\frac{\begin{array}{c}\phi_1; \Delta_1 \models \exists\phi.\Delta \quad \phi_1; \phi; \Delta; \Gamma \vdash e_1 : (\phi_2; \Delta_2; \texttt{bool}(i)) \\ \phi_2, i = 1; \Delta_2; \Gamma \vdash e_2 : (\phi_3; \Delta_3; \texttt{unit}) \quad \phi_3; \Delta_3 \models \exists\phi.\Delta\end{array}}{\phi_1; \Delta_1; \Gamma \vdash \mathbf{while}(e_1, e_2) : (\phi_2, i = 0; \Delta_2; \texttt{unit})} \textbf{ (type-while)}$$

$$\frac{\phi_1; \Delta_1; \Gamma \vdash e_1 : (\phi_2; \Delta_2; \tau_1) \quad \phi_2; \Delta_2; \Gamma, X : \tau_1 \vdash e_2 : (\phi_3; \Delta_3; \tau_2)}{\phi_1; \Delta; \Gamma \vdash \mathbf{let}\ X = e_1\ \mathbf{in}\ e_2\ \mathbf{end} : (\phi_3; \Delta_3; \tau_2)} \textbf{ (type-let)}$$

$$\frac{\phi_1 \vdash i : \gamma \quad \phi_1; \Delta_1; \Gamma \vdash e : (\phi_2; \Delta_2; \tau)}{\phi_1; \Delta_1; \Gamma \vdash \langle i \mid e \rangle : (\phi_2; \Delta_2; \Sigma a : \gamma.\tau)} \textbf{ (type-pack)}$$

$$\frac{\begin{array}{c}\phi_1; \Delta_1; \Gamma \vdash e_1 : (\phi_2; \Delta_2; \Sigma a : \gamma.\tau_1) \\ \phi_2, a : \gamma; \Delta_2; \Gamma, X : \tau_1 \vdash e_2 : (\phi_3; \Delta_3; \tau_2)\end{array}}{\phi_1; \Delta; \Gamma \vdash \mathbf{unpack}\ e_1\ \mathbf{as}\ \langle a \mid X \rangle\ \mathbf{in}\ e_2\ \mathbf{end} : (\phi_3; \Delta_3; \tau_2)} \textbf{ (type-unpack)}$$

Figure 10. Some typing rules for $\text{Xanadu}_0^{\Pi,\Sigma}$

$\texttt{int}(2)$. In other words, type equality is modulo constraint satisfaction.

We write $\phi \models \tau_1 = \tau_2$ to mean that types τ_1 and τ_2 are equivalent under the index variable context ϕ. Similarly, we write $\phi \models \tau_1 \leq \tau_2$ to mean that type τ_1 coerces into type τ_2 under ϕ. The rules for type equality and coercion are presented in Figure 12.

Notice that the need for deriving judgments of forms $\phi \vdash \theta : \phi'$, $\phi \models \tau_1 = \tau_2$ and $\phi; \Delta \models \exists\phi'.\Delta'$ involves constraint satisfaction.

3.4 Type Soundness

We now state the type soundness theorem for $\text{Xanadu}_0^{\Pi,\Sigma}$ as follows.

Theorem 3.1 *Let* $P = \mathbf{letdef}\ D\ \mathbf{in}\ e\ \mathbf{end}$ *be a program such that* $\vdash P$ *is derivable; if* $\mathcal{M}[e] \to_D^* \mathcal{M}'[e']$*, then either* e' *is* $\langle\rangle$ *or* $\mathcal{M}'[e'] \to_D \mathcal{M}''[e'']$ *for some* $\mathcal{M}''$ *and* e''*. In other words, the evaluation of a well-type program in* Xanadu_0 *either terminates normally or runs forever.*

The theorem can be proven by following the same approach as is used in the proof for Theorem 2.3, though it is much more involved this time. Please find more details in [12].

4 Type Inference

We have so far presented an implicitly typed language $\text{Xanadu}_0^{\Pi,\Sigma}$. The typing rules in $\text{Xanadu}_0^{\Pi,\Sigma}$ are *not* syntax-directed, making it difficult to implement a practical type inference algorithm for programs written in $\text{Xanadu}_0^{\Pi,\Sigma}$. Therefore, it becomes necessary to provide an external language in which the programmer can supply type annotations

$$\frac{\mathcal{M}[e] \rightarrow_D \mathcal{M}'[e']}{\mathcal{M}[\langle i \mid e \rangle] \rightarrow_D \mathcal{M}'[\langle i \mid e' \rangle]} \ \textbf{(eval-pack)}$$

$$\frac{\mathcal{M}[e_1] \rightarrow_D \mathcal{M}'[e_1']}{\mathcal{M}[\textbf{unpack } e_1 \textbf{ as } \langle a \mid X \rangle \textbf{ in } e_2 \textbf{ end}] \rightarrow_D \mathcal{M}'[\textbf{unpack } e_1' \textbf{ as } \langle a \mid X \rangle \textbf{ in } e_2 \textbf{ end}]} \ \textbf{(eval-unpack-1)}$$

$$\frac{}{\mathcal{M}[\textbf{unpack } \langle i \mid hc \rangle \textbf{ as } \langle a \mid X \rangle \textbf{ in } e_2 \textbf{ end}] \rightarrow_D \mathcal{M}[e_2\{a \mapsto i\}\{X \mapsto hc\}]} \ \textbf{(eval-unpack-2)}$$

Figure 11. Additional evaluation rules for $\mathrm{Xanadu}_0^{\Pi,\Sigma}$

$$\frac{\phi \models x = y}{\phi \models \mathtt{int}(x) = \mathtt{int}(y)} \ \textbf{(eq-int)}$$

$$\frac{\phi \models \tau_1 = \tau_2 \quad \phi \models x = y}{\phi \models \tau_1 \ \mathtt{array}(x) = \tau_2 \ \mathtt{array}(y)} \ \textbf{(eq-array)}$$

$$\frac{\phi, a : \gamma \models \tau_1 = \tau_2}{\phi \models \Sigma a : \gamma.\tau_1 = \Sigma a : \gamma.\tau_2} \ \textbf{(eq-exi-ivar)}$$

$$\frac{\phi \models \tau_1 = \tau_2}{\phi \models \tau_1 \leq \tau_2} \ \textbf{(co-eq)}$$

$$\frac{}{\phi \models \tau \leq \mathtt{top}} \ \textbf{(co-top)}$$

$$\frac{\begin{array}{c}\phi \models \Delta(x) \leq \Delta'(x) \\ \text{for all } x \in \mathbf{dom}(\Delta) = \mathbf{dom}(\Delta')\end{array}}{\phi; \Delta \models \Delta'} \ \textbf{(co-context)}$$

$$\frac{\phi \vdash \theta : \phi' \quad \phi; \Delta \models \Delta'[\theta]}{\phi; \Delta \models \exists \phi'.\Delta'} \ \textbf{(co-state-type)}$$

Figure 12. Some type equality and coercion rules for $\mathrm{Xanadu}_0^{\Pi,\Sigma}$

to facilitate type inference. We outline some key decisions we have made in the design of an external language *Xanadu*.

We omit most details on the design of a type inference algorithm for Xanadu, which largely follows the bi-directional approach explained in [16]. However, we explain a key step in type inference that involves synthesizing state type invariant for both loops and conditionals.

4.1 Master Types for Variables

In theory, a reference variable in $\mathrm{Xanadu}_0^{\Pi,\Sigma}$ is allowed to store a value of any type. However, we impose some restriction on this feature in practice. We assign every reference variable x a type τ and allow a value to be stored in x only if the value can be coerced to have type τ. We call τ the master type of x and write $\mu(x)$ for it.

For the implementation of binary search in Figure 1, the master type of `low` is `int[0,n]`, which indicates that `low` can only store an integer whose value is between `0` and `n`. Similarly, the master type of `high` is `int[-1,n)`, meaning that only an integer whose value j satisfying $-1 \leq j < n$ can be stored in `high`. The master types of variables `mid` and `x` are `int`, that is, these variables can only store integers. For a variable appearing as an argument in a function declaration, the master type of the variable is assumed to be the type erasure of the type of the argument (unless the programmer declare the master type of the variable explicitly). For instance, the master types of `key` and `vec` are `int` and `int array`, respectively.

4.2 No Value Variables

Value variables are not available to the programmer in our current implementation. The need for value variables occurs in the elaboration phase where a program is transformed from external representation into internal representation and then type-checked. We present a simple example to illustrate this point.

Suppose that the assignment `x2 = fact(x1) + 1` occurs in a program, where function `fact` has already been given the type $\tau = (\mathtt{int}) \rightarrow \mathtt{int}$. This assignment is formally represented as $x_2 := +(\mathbf{call}(Fact; x_1), 1)$, where $Fact$ is declare to have type τ. Unfortunately, this assignment does not type check since the type of $+$ is $\Pi\{i : nat, j : nat\}.(\mathtt{int}(i), \mathtt{int}(j)) \rightarrow \mathtt{int}(i + j)$ and it is impossible to coerce `int` into $\mathtt{int}(i)$ for any index i. In order to overcome the problem, we elaborate `x2 = fact(x1) + 1` into $x_2 := \textbf{unpack } \mathbf{call}(Fact; x_1) \textbf{ as } \langle a \mid X \rangle \textbf{ in } +(X, 1) \textbf{ end}$, which can be readily typed.

4.3 State Type Invariant Synthesis

It is readily seen that we need a state type $\exists \phi.\Delta$ in order to type either a conditional $\mathbf{if}(e, e_1, e_2)$ or a loop $\mathbf{while}(e_1, e_2)$. Clearly, such a state type, which is essentially an invariant about the types of some reference variables at a program point, must be supplied by the programmer or automatically synthesized. This is to be a crucial issue in the design of a type inference algorithm for Xanadu.

A state type invariant for a loop is of form $\exists\phi.\Delta$. It would obviously be too obtrusive if the programmer had to write a loop invariant for each loop. Therefore, it is important in practice to effectively synthesize loop invariants for common cases. We are less enthusiastic about sophisticated approaches to loop invariant synthesis since such approaches are usually highly heuristic and often make it exceedingly difficult for the programmer to figure out the cause of type errors in case they occur.

We use the example in Figure 1 to illustrate the simple approach we have adopted for loop invariant synthesis in our current implementation of Xanadu.

Let (ϕ, Δ_0) be the invariant hint provided by the programmer at the beginning of a loop. If there is no hint provided, we assume $(\phi, \Delta_0) = (\cdot, \cdot)$, that is, both ϕ and Δ_0 are empty. Also, let $\tau(x)$ be the type of the reference variable x immediately before loop entrance.

- We first list all the variables $x_1, \ldots, x_n$ in the loop whose values may potentially be modified during the execution of the loop at run-time. For the loop in Figure 1, the variables `x, mid, low, high` belong to such a list, but the variables `key, vec` do not.

- Let Δ_1 be a reference variable context whose domain consists of all declared reference variables and

$$\Delta_1(x) = \begin{cases} \Delta_0(x) & \text{if } x \in \mathbf{dom}(\Delta_0); \\ \mu(x) & \text{if } x = x_i \notin \Delta_0 \text{ and } \tau(x) \neq \texttt{top}; \\ \tau(x) & \text{otherwise.} \end{cases}$$

  For the loop in Figure 1, Δ_1 maps `x, mid, low, high` to the following types, respectively: `top`, `top`, `int[0,n]`, and `int[-1,n)`. In addition, Δ_1 maps `key` and `vec` to `int` and `int array(n)`, respectively.

- We use $\exists\phi.\Delta_1$ for the loop invariant. For the loop in Figure 1, it can be verified that $\exists\phi.\Delta_1$ is indeed a loop invariant and this invariant suffices to guarantee that the array subscripting `vec[mid]` in the loop is safe, that is, `mid` is within the bounds of `vec`.

The state type invariant synthesis for a conditional is done similarly.

4.4 The External Language Xanadu

The syntax for Xanadu is given in Figure 13. The expression $\lambda\phi.(\Delta, \mathbf{while}(e_1, e_2))$ conveys that (ϕ, Δ) is the hint mentioned in Section 4.3 that facilitates state type invariant synthesis for the loop $\mathbf{while}(e_1, e_2)$. Note that $\lambda\phi$ is used to indicate that this expression is polymorphic on ϕ, that is, what is executed at run-time is $\mathbf{while}(e_1[\theta], e_2[\theta])$ for some substitution θ with "type" ϕ. The expression $\lambda\phi.(\Delta, \mathbf{if}(e, e_1, e_2))$ is interpreted similarly. In $\mathbf{newvar}\ x : \tau\ \mathbf{in}\ e\ \mathbf{end}$, τ is the master type of the newly declared variable x.

$$\begin{array}{l} \text{expressions} \quad e \ ::= \\ \quad c \mid x \mid op(e_1, \ldots, e_n) \mid x := e \mid e_1; e_2 \mid \\ \quad \lambda\phi.(\Delta, \mathbf{if}(e, e_1, e_2)) \mid \lambda\phi.(\Delta, \mathbf{while}(e_1, e_2)) \mid \\ \quad \mathbf{alloc}(e_1, e_2) \mid \mathbf{arraysize}\ (e) \mid \\ \quad e_1[e_2] \mid e[e_1] := e_2 \mid \\ \quad \mathbf{newvar}\ x : \tau\ \mathbf{in}\ e\ \mathbf{end} \mid \mathbf{call}(X; e_1, \ldots, e_n) \end{array}$$

Figure 13. The syntax for external language

It should be clear how the presented examples in concrete syntax such as the one in Figure 1 can be mapped into the corresponding expressions in the formal syntax for Xanadu.

4.5 An Example

We briefly explain how the array subscripting `vec[mid]` in Figure 1 is proven safe in the type system of Xanadu.

The presented type invariant synthesis approach yields the state type $\exists\phi.\Delta$ for the loop in Figure 1, where ϕ is empty and Δ declares that the variables `low`, `high` and `mid` and `x` have types `int[0,n]`, `int[-1,n)`, `top` and `top`. We apply the rule **(type-open)** to `low` and `high`. Then `low` has type $\texttt{int}(i)$ for index variable i of sort $\{a : int \mid 0 \leq a \leq n\}$, where index variable n is declared of sort nat. Similarly, `high` has type $\texttt{int}(j)$ for index variable j of sort $\{a : int \mid -1 \leq a < n\}$. After the assignment `mid = (low + high) / 2`, `mid` has type $\texttt{int}((i+j)/2)$. We now need to prove $0 \leq (i+j)/2 < n$ under the assumption that n is a natural number, $0 \leq i \leq n$, $-1 \leq j < n$ and $i \leq j$. This can be readily verified. Note that $i \leq j$ is in the assumption because the loop condition must hold when the assignment is `x = vec[mid]` executed.

5 Extensions

We now describe some programming features that can be incorporated into Xanadu to make it a more realistic language. Note that *all* these features are already supported in a prototype implementation of Xanadu.

5.1 Tuples and Polymorphism

Tuples, which are not present in $\text{Xanadu}_0^{\Pi,\Sigma}$ for the sake of a less involved presentation, can be readily added into Xanadu.

The interaction between the dependent type system of $\text{Xanadu}_0^{\Pi,\Sigma}$ and polymorphism is minor. As demonstrated in Figure 2, polymorphism is already included in Xanadu. As a matter of fact, even polymorphic recursion is available in Xanadu. The type of every declared function in Xanadu is provided by the programmer and therefore it adds virtually no cost to support polymorphic recursion.

5.2 Higher-order Functions

Currently, we allow a function to accept functions as its arguments but forbid a function call to return a function. The type system of Xanadu could be readily extended to support curried functions but such an extension would make it greatly more complicated to compile Xanadu programs. It should be further studied whether it is worth the effort to fully support higher-order functions in Xanadu.

5.3 Exceptions

The exception mechanism similar to the one in Java can be readily incorporated into Xanadu. We currently support exceptions like **break** and **continue** in a loop to allow for altering the control-flow. Also we allow the use of **return**(e) to immediately return the evaluation result of e to the caller of a function and the use of **exit** to abnormally stop the evaluation of a program.

5.4 Union Types and Record Types

A mechanism for declaring dependent union types, which directly corresponds to dependent datatypes in DML [16, 8] is added into Xanadu and pattern matching is provided for decomposing the values of a union type. A concrete example of this feature is given in Figure 2.

An example of dependent record type is given in Figure 3, where the type `<'a>sparseArray(m,n)` is declared for sparse arrays of dimension `m` by `n` in which all elements are of type `'a`.

5.5 Global Variables

It is also allowed to declare global variables in Xanadu. The inclusion of global variables leads to some difficulty. In the following example, a global variable `count` is declared and initialized with 0. It is required that every global variable be initialized upon declaration. The master type of `count` is `[i:nat] int(i)`, meaning that only a natural number can be stored in `count`.

```
global count: [i:nat] int(i) = 0
```

Some implications from adding global variables are explained in [12] and approaches to addressing these implications are also given there.

6 Implementation

We have prototyped a type-checker for Xanadu that handles all the features mentioned in this paper and a rudimentary compiler for compiling a Xanadu program into a form that is then interpreted. The implementation is written in Objective Caml and its current version is available online at [13], where one can also find many running program examples in Xanadu, including implementations of binary search, fast Fourier transform, heapsort, Gaussian elimination, red-black trees, random-access lists, various list functions, etc. The type-checker consist of two phases in which the first one checks whether the erasure[4] of a Xanadu program is well-typed and the second one performs dependent type-checking. The first phase is straight forward and the second phase involves solving linear constraints on integers. The method used in the implementation for solving linear constraints is based on Fourier-Motzkin elimination [1] and some information on this is already mentioned in [15].

7 Related Work

Generally speaking, there are two directions for extending a Hindely-Milner style of type system. One direction is to extend it so as to accept more programs as type-correct and the other is to extend it so as to assign more accurate types to programs. Our work follows the second direction. A pioneering study in this direction is the work on refinement types [3], which aims at expressing and checking more properties of programs that are already well-typed in ML, rather than admitting more programs as type-correct, which is the goal of most other research on extending type systems. The mechanism of refinement types incorporates the notion of intersection types and can thus ascribe multiple types to terms in a uniform way.

DML is a functional programming language that enriches ML with a restricted form of dependent types [16], allowing the programmer to capture more program invariants through types and thus detect more program errors at compile-time. In particular, the programmer can refine datatypes with type index expressions in DML, capturing more invariants in various data structures. For instance, one can form a datatype in DML that is precisely for all red/black trees and do practical programming with such a type. The type system of DML is also studied for array bound check elimination [15].

Most closely related to DML is the system of *indexed types* developed independently by Zenger in his Ph.D. Thesis [18] (an earlier version of which is described in [17]). He works in the context of lazy functional programming. In general, his approach seems to require more changes to a given Haskell program to make it amenable to checking indexed types than is the case for DML and ML. This is particularly apparent in the case of existential dependent types, which are tied to data constructors. This has the advantage of a simpler algorithm for elaboration and type-checking than ours, but the program (and not just the type) has to be more explicit.

[4] By erasure we basically mean ignoring all syntax related to type index expressions.

Typed Assembly Language (TAL) is introduced in [5], in which a significant feature is that the type of a register is allowed to change during execution. This sheds some light on the design of Xanadu. However, a language like Xanadu_0, which essentially adopts the notion of TAL at the source level seems of limited interest.

A dependently typed assembly language (DTAL) is designed on top of TAL with a dependent type system to overcome some limitations inherent in TAL [14]. The type system of DTAL is capable of capturing memory safety, which includes both type safety and safe array subscripting. The notion of dependent types in DTAL is adopted from DML. The design of Xanadu is partly prompted by the need for generating DTAL code. Also, the techniques employed in establishing the type soundness for $\text{Xanadu}_0^{\Pi,\Sigma}$ are largely adopted from [14].

The work on extended static checking (ESC) [2] also emphasizes the use of formal annotations in capturing the program invariants. These invariants can then be verified through (light-weight) theorem proving. ESC is developed on top of the imperative programming language Modular-3, taking an approach based on first-order logic assertions. It provides a specification language for the programmer to specify properties including a list of variables that a procedure may modify, a precondition which must be satisfied before a function call, a postcondition that must hold when a function terminates, and so forth. Further study is needed to determine whether ESC can readily handle higher-order functions.

8 Conclusion

We have presented in the design of Xanadu a novel approach to enriching imperative programming with a form of dependent types. This includes forming both static and dynamic semantics for Xanadu and then establishing its type soundness. We have also prototyped a type-checker for Xanadu and a rudimentary compiler for compiling a Xanadu program into a form that is then interpreted, demonstrating a proof of concept.

In future work, we plan to continue the development of Xanadu, extending the language with features such as inner functions and modules. Also we intend to study the use of dependent types in compilation, compiling Xanadu into DTAL.

9 Acknowledgment

I thank Jerry Paul for reading a draft and providing me with his comments.

References

[1] G. Dantzig and B. Eaves. Fourier-Motzkin elimination and its dual. *Journal of Combinatorial Theory (A)*, 14:288–297, 1973.

[2] D. Detlefs. An overview of the extended static checking system. In *Workshop on Formal Methods in Software Practice*, 1996.

[3] T. Freeman and F. Pfenning. Refinement types for ML. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 268–277, Toronto, Ontario, 1991.

[4] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, Naples, Italy, 1984.

[5] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[6] G. Necula. Proof-carrying code. In *Conference Record of 24th Annual ACM Symposium on Principles of Programming Languages*, pages 106–119, Paris, France, 1997. ACM press.

[7] G. Necula and P. Lee. The design and implementation of a certifying compiler. In *ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 333–344. ACM press, June 1998.

[8] H. Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, 1998. pp. viii+189. Available as `http://www.cs.cmu.edu/~hwxi/DML/thesis.ps`.

[9] H. Xi. Dead code elimination through dependent types. In *The First International Workshop on Practical Aspects of Declarative Languages*, San Antonio, January 1999.

[10] H. Xi. Some Practical Aspects of Dependent Datatypes, November 1999. Available as `http://www.cs.bu.edu/~hwxi/academic/papers/PADD.ps`.

[11] H. Xi. Facilitating Program Verification with Dependent Types, March 2000. Available as `http://www.cs.bu.edu/~hwxi/academic/papers/FPVDT.ps`.

[12] H. Xi. Imperative Programming with Dependent Types. In *Proceedings of 15th IEEE Symposium on Logic in Computer Science*, pages 375–387, Santo Barbara, June 2000.

[13] H. Xi. Xanadu: Imperative Programming with Dependent Types, 2001. Available at `http://www.cs.bu.edu/~hwxi/Xanadu/`.

[14] H. Xi and R. Harper. A Dependently Typed Assembly Language. Technical Report CSE-99-008, Oregon Graduate Institute, July 1999. Also available as `http://www.cs.bu.edu/~hwxi/academic/papers/DTAL.ps`.

[15] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 249–257, Montréal, Canada, June 1998.

[16] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, Texas, January 1999.

[17] C. Zenger. Indexed types. *Theoretical Computer Science*, 187:147–165, 1997.

[18] C. Zenger. *Indizierte Typen*. PhD thesis, Fakultät für Informatik, Universität Karlsruhe, 1998.