June 2000

University of Cincinnati

Hongwei X.

Compiling with Dependent Types
from high-level dependent types in source programs to low-level dependent types are to be translated.

Self-certification is done through the use of a form of

To produce (optimised) self-certified code that can

What do we want to do?
The programmer may need to provide dependent type annotations for type-checking purpose during execution.

The type of a variable in 
Xanthu can change.

Xanthu (X', 2000) is a dependently typed imperative language like C-like syntax.

Source Language: Xanthu
return sum

\[
\sum_{i=0}^{n-1} a[i] \cdot b[i] = \sum
\]

for (i = 0; i < \|a\|; i = i + 1)
sum = 0.0;

/* nat is [a:int | a >= 0 [int] e] */

var: nat \|\ \| float sum: float dotprod (ten: int (n), a:nat: float, b[nat: float])

\[
\text{A Program in Xanadu}
\]
A polymorphic datatype for \texttt{List}:

\begin{verbatim}
{ cons (n+1) of \langle a \rightarrow \texttt{List}\langle n \rangle \rangle,
  \texttt{Nil} (0),
}
union \langle a \rightarrow \texttt{List} \rangle\texttt{with nat} =
\end{verbatim}

\textit{A Datatype in Xanthu}
Another Program in Xandu
1999) for certifying memory safety
dent types adopted from DML (X! and Penning)
(Morisset et al., 1998) with a form of depend-
TAL was originally designed to extend TAL

(X! and Harper, 1999)
TAL is a dependently typed assembly language

Target Language: TAL
In DTA, information is transformed into a state type,
- We also briefly explain how a state type annotates
- In particular, we show how dependent datatypes
- This talk is focused on low-level data representation

What is in this talk?
\{ u > v \mid v \geq 0 \text{ int} : \text{nat} \}

where \text{nat} \Downarrow

\begin{align*}
\text{nat' } \Downarrow \quad & u + \cdots + 1 \\
\text{nat'' } \Downarrow \quad & (u, 1) \\
\text{nat" } \Downarrow \quad & (u, 1) \\
\text{nat^3 } \Downarrow \quad & (u, 1) \\
\text{int } \Downarrow \quad & \text{int} \\
\text{float } \Downarrow \quad & \text{float} \\
\text{top } \Downarrow \quad & \text{top}
\end{align*}

\text{Low-level Types}
\begin{align*}
\max(\mid u \mid, \ldots, \mid \top \mid) &= \mid (u, \ldots, \top, \top) \delta \mid \\
\mid u \mid + \cdots + \mid \top \mid &= \mid (u, \ldots, \top, \top) \lambda \mid \\
\top &= \mid (\bot, p) \psi \mid \\
\bot &= \mid \bot \mid \\
\top &= \mid u \mid
\end{align*}
\[ \langle t_1, \ldots, t_n, \tau \rangle \text{ is represented flatly as} \]

| \( t_n \) | \( \ldots \) | \( \tau \) |
8, \( p \), is represented as

\[ \text{Pointer Type Representation} \]
An Example of Boxing

The high-level type `int * float` may be translated into the low-level type:

```
&((0, T(int, &((0, float))))
```
A tag function for \((\tau_1, \ldots, \tau_m)\) has the following type:

\[
\Pi a : \mathsf{nat} \rightarrow \delta(a; \tau_1, \ldots, \tau_m) \rightarrow \mathsf{int}(a)
\]
\[
\text{Let } T_1 = \mathcal{G}(1, T(\text{int}(0), \text{int}(1)), \text{int}) \text{ and } T_2 = \mathcal{G}(1, T(\text{int}(1), \text{float}(T)))
\]

Then the following is a tag function for \((T_1, T_2)\):

\[
\text{tag (x) = (ret y, load y, x \rightarrow y)}; \text{ var y;}
\]

\text{ret y; load y, x \rightarrow y; var y;}

\text{An Example of Tag Function}
An Example of Tagging (I)
An Example of Tagging (II)

\[ \Sigma a : nat_2. \delta(a; \&(1, T(int(0), int)), \&(1, T(int(1), float))) : \]

| int(0)/int(1) | int/float |
even if we could not find a tag function for $(T, \ldots, T^n)$?

Would it still be useful to form an interesting question?
where $T_1 = \text{int}$ and $T_2 = \mathbb{R}(0, \text{float})$.

$\exists a : \text{natn} \cdot \text{int}(a) \cdot g(a, T_1, T_2)$

low-level type:

The high-level type $\text{int} + \text{float}$ may also be translated into the

Another Example of Tagging
Recursive Sum Type (I)

We have:

\( \langle \alpha \rangle \text{list} = \Sigma a : \text{nat} \cdot \langle \alpha \rangle \text{list} + a \cdot \langle \alpha \rangle \text{list} \)

\( \langle \alpha \rangle \text{list} \) may be translated into \( \langle \alpha \rangle \text{list} \).
a null pointer. If we can distinguish a null pointer from one that is not
where \( \alpha(T) = (\alpha(T) \oplus (0; \text{top}(0))). \)

\[
\begin{align*}
(\alpha(T) \oplus (0; \text{top}(0))) &\equiv \alpha(T) \\
\text{nat}_{2; \alpha(T) \oplus (0; \text{top}(0))} &\equiv \text{nat}_{2; \alpha(T)} \\
\alpha(T \oplus (0; \text{top}(0))) &\equiv \alpha(T) \oplus (0; \text{top}(0)) \\
\text{nat}_{2; \alpha(T) \oplus (0; \text{top}(0))} &\equiv \text{nat}_{2; \alpha(T) \oplus (0; \text{top}(0))} \\
\alpha(T \oplus (0; \text{top}(0))) &\equiv \alpha(T) \oplus (0; \text{top}(0)) \\
\text{nat}_{2; \alpha(T) \oplus (0; \text{top}(0))} &\equiv \text{nat}_{2; \alpha(T) \oplus (0; \text{top}(0))}
\end{align*}
\]

We have

\[
(II)
\]

Recursive Sum Type (II)
Dependent Datatypes (I)
\[ \text{where } \phi \text{ is } a + 1. \]

\[ \text{We have } \]

**Dependent Datatypes (II)**
dependent datatypes (III)

may be translated into \((\alpha(\text{list}(a)))\).

where \(\phi \in \text{nat} \vdash 0 = n \land \phi \therefore \text{nat}, u : \text{nat}, n + 1 = a + 1.

\text{let list} =
25

```
{ }

{ }

\text{case Cons} \ (\text{`}, \ \text{xs}) \ : \ x = x + 1; \ \\
\text{case Nil} \ : \ \text{return} \ x; \ \\
\} \ (\text{xs}) \ \\
\} \ (\text{true}) \ \\
\text{invariant} : \ [\text{t}, \text{n} : \text{nat} | \text{t} + 1 = n] \ (\text{xs} : \forall a, \ \text{List} (\text{t}), x : \text{Int} (\text{t})) \)
```

We now explain how the following code fragment is compiled.

---

A Code Fragment in Xanadu
A Code Fragment in DTLA

```plaintext
jump loop
add r2, r2, i
load rt, r1(t)
beq r3, finish
load r3, r1(-t)

/* unfold rt */ for type-checking purpose */

[ x' : [ x', :: x', :: sp: int(t), sp: test(t), rt: int(f) ] ]

{ r1 : a, r2 : a } ( r1 : nat, i : nat, j : nat | i + j = n )

loop : a, a
```
In particular, would it be beneficial to introduce the notion of macros? How?

Would it be beneficial to introduce a low-level language with some structures instead of harnessing native code directly?
Conclusion

Translating (dependent) types from source level to target level presents an effective approach to establishing properties on low-level code, which could often be difficult to prove from scratch.

This practice can also be of great help for debugging a compiler.