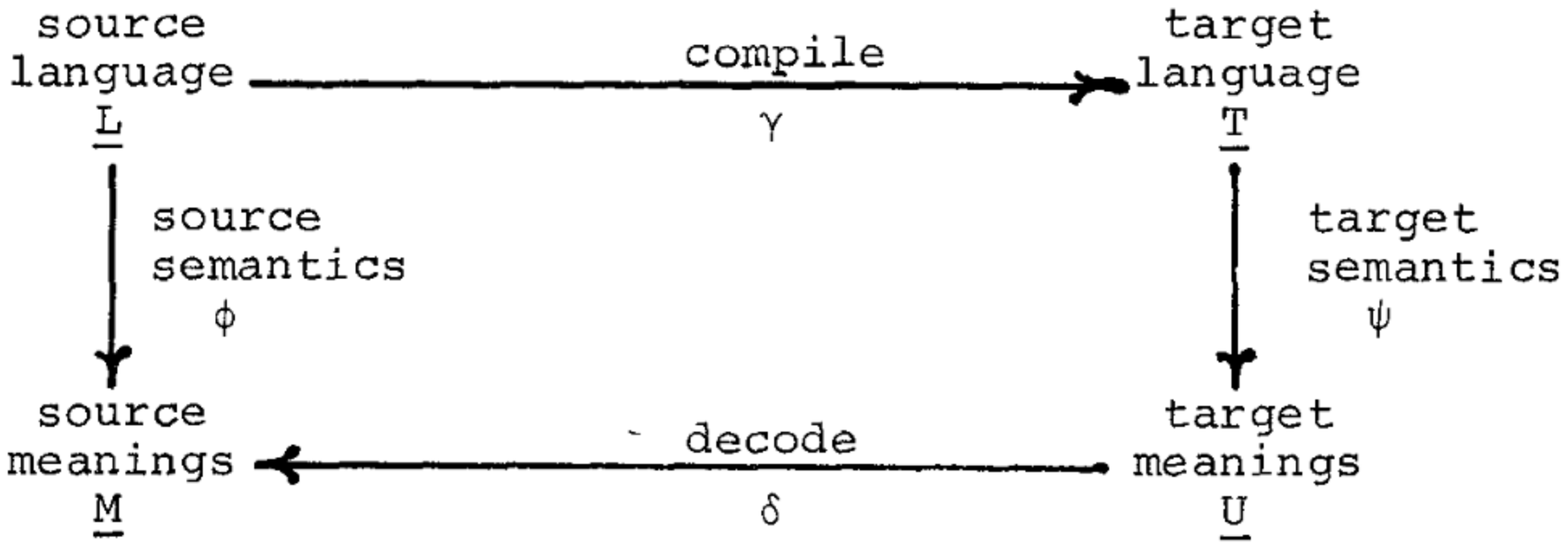
# DimSum: A Decentralized Approach to Multi-language Semantics and Verification

Michael Sammler, Simon Spies,
Youngju Song, Emanuele D'Osualdo,
Robbert Krebbers, Deepak Garg, Derek Dreyer

**POPL 2023 Distinguished Paper**

**Presented by Priya Srikumar, at Cornell University PLDG, Spring 2023**

# Program verification needs compiler correctness!




F. Lockwood Morris. 1973. Advice on structuring compilers and proving them correct.
In Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL '73).
Association for Computing Machinery, New York, NY, USA, 144–152. https://doi.org/10.1145/512927.512941

# …but compiling whole programs isn’t realistic

```
int src = 1;
int dst;

asm ("mov %1, %0\n\t"
     "add $1, %0"
     : "=r" (dst)
     : "r" (src));

printf("%d\n", dst);
```

# Verifying multi-language programs is hard (!)

- Languages have different interfaces/data representations
- Prior work limits the structure of multi-language programs
  - Fixed source language
  - Fixed set of languages
  - Fixed memory model
  - Fixed interoperation mechanisms

# DimSum employs a *decentralized* approach

- Framework to specify/verify multi-language programs
- Notion of compiler correctness via module refinement
- Language-agnostic combinators to link/translate modules
- Users define labeled transition systems over language events
- Users define embeddings between each pair of languages

# Case study: `Asm` and `Rec`

***Trace events***

- `Jump` and `Syscall`
- `Call` and (function) `Return`
- ! for an outgoing event
  - `Syscall!(…)` is an outgoing syscall
- ? for an incoming event
  - `Return?(…)` is an incoming function return

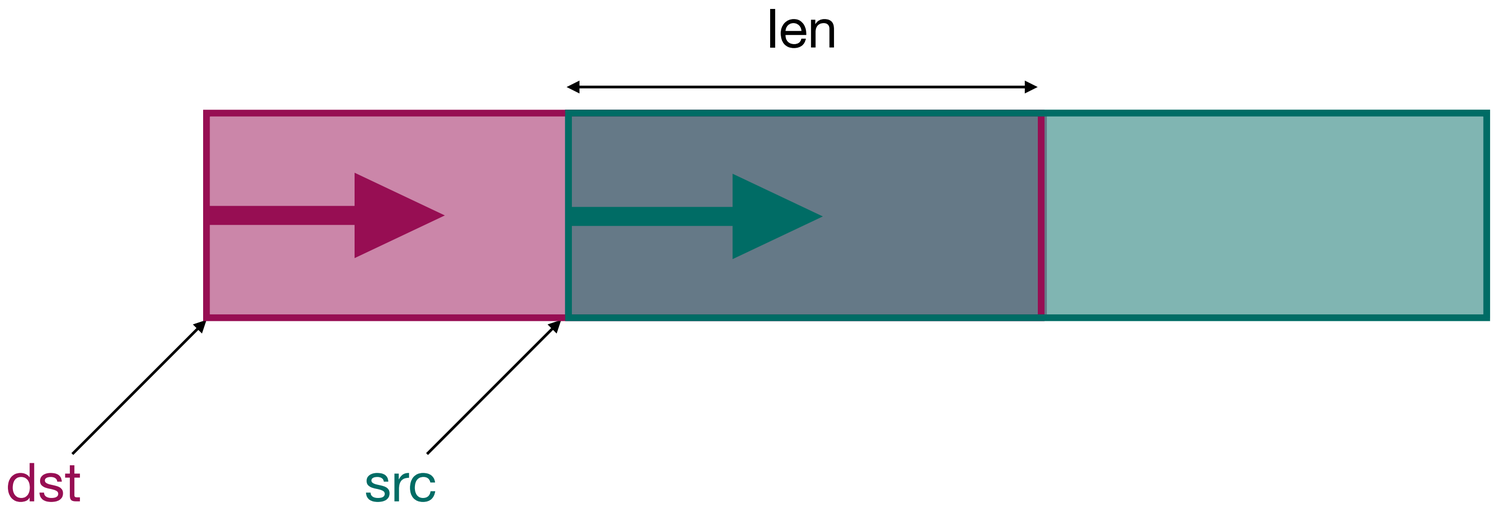
# main

```
fn main() {
  local x[3];
  x[0] ← 1;
  x[1] ← 2;
  // x ↦ [1, 2, 0]
  memmove(x + 1, x + 0, 2);
  // x ↦ [1, 1, 2]
  print(x[1]); print(x[2])
  // print 1, then 2
}
```

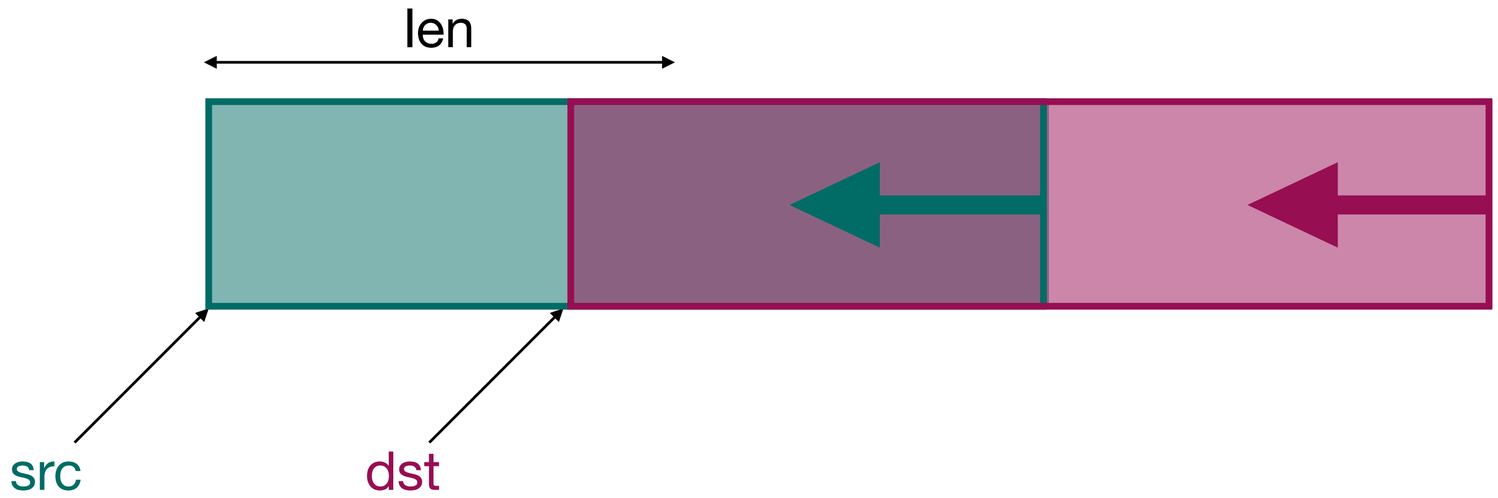
# main

```
fn main() {
  local x[3];
  x[0] ← 1;
  x[1] ← 2;
  // x ↦ [1, 2, 0]
  memmove(x + 1, x + 0, 2);
  // x ↦ [1, 1, 2]
  print(x[1]); print(x[2])
  // print 1, then 2
}
```

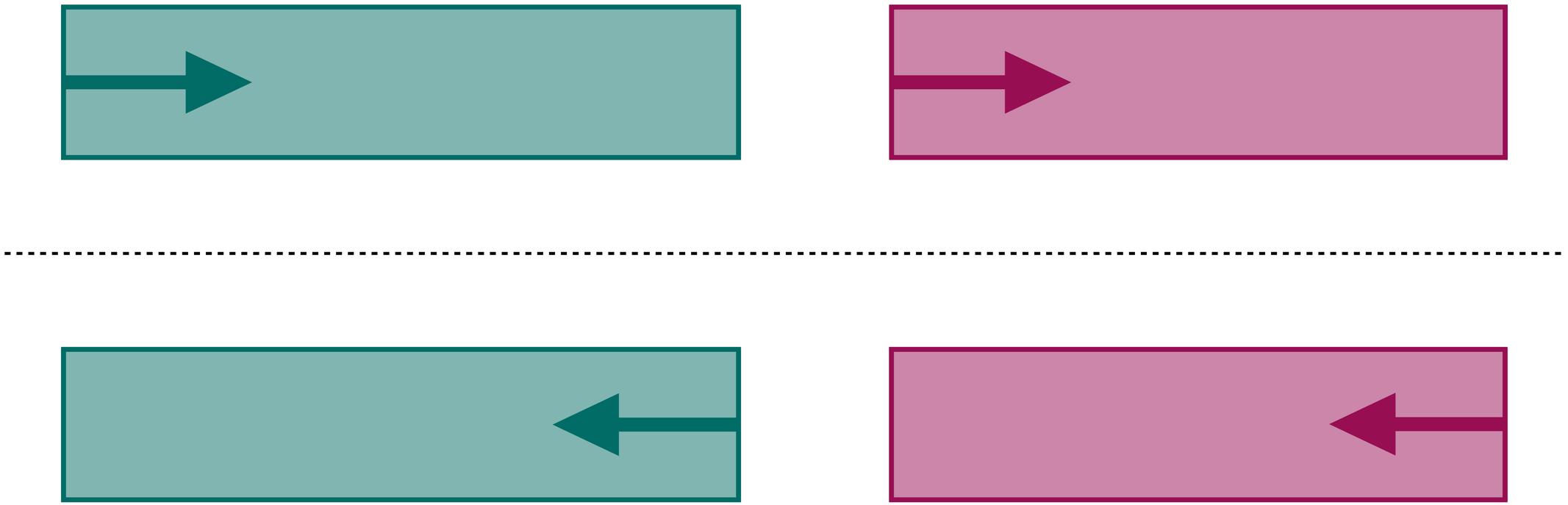
# Correctness of `memmove(dst, src, len)`



# Correctness of `memmove(dst, src, len)`



# Correctness of `memmove(dst, src, len)`



# memmove

```
fn memmove(d, s, n) {
 if locle(d, s)
 then
  memcpy(d, s, n, 1)
 else
  memcpy(d + n - 1, s + n - 1, n, -1)
}
```

# memmove

```
fn memmove(d, s, n) {
 if locle(d, s)
 then
  memcpy(d, s, n, 1)
 else
  memcpy(d + n - 1, s + n - 1, n, -1)
}
```

# locle

```
locle : sle x0, x0, x1;
        ret
```

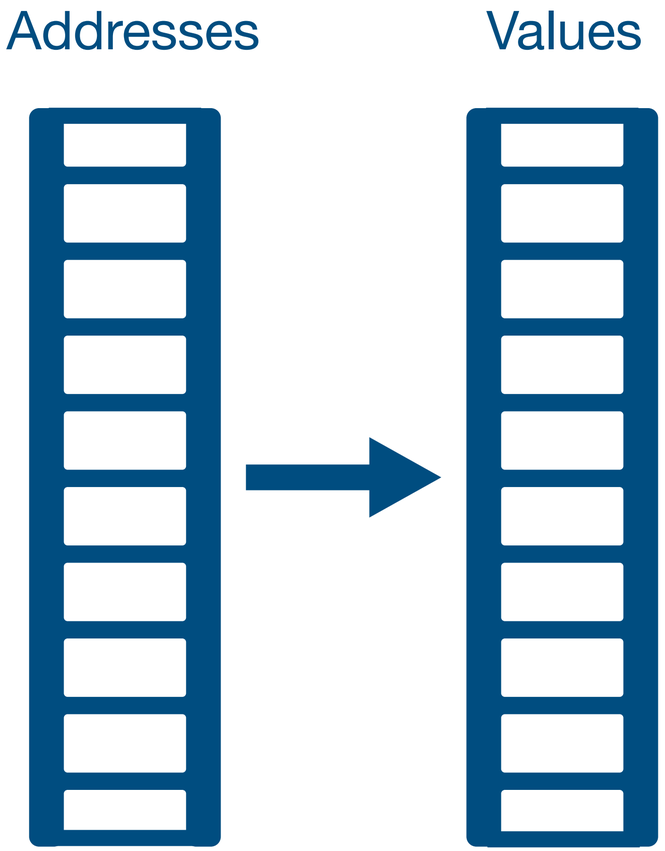
# locle

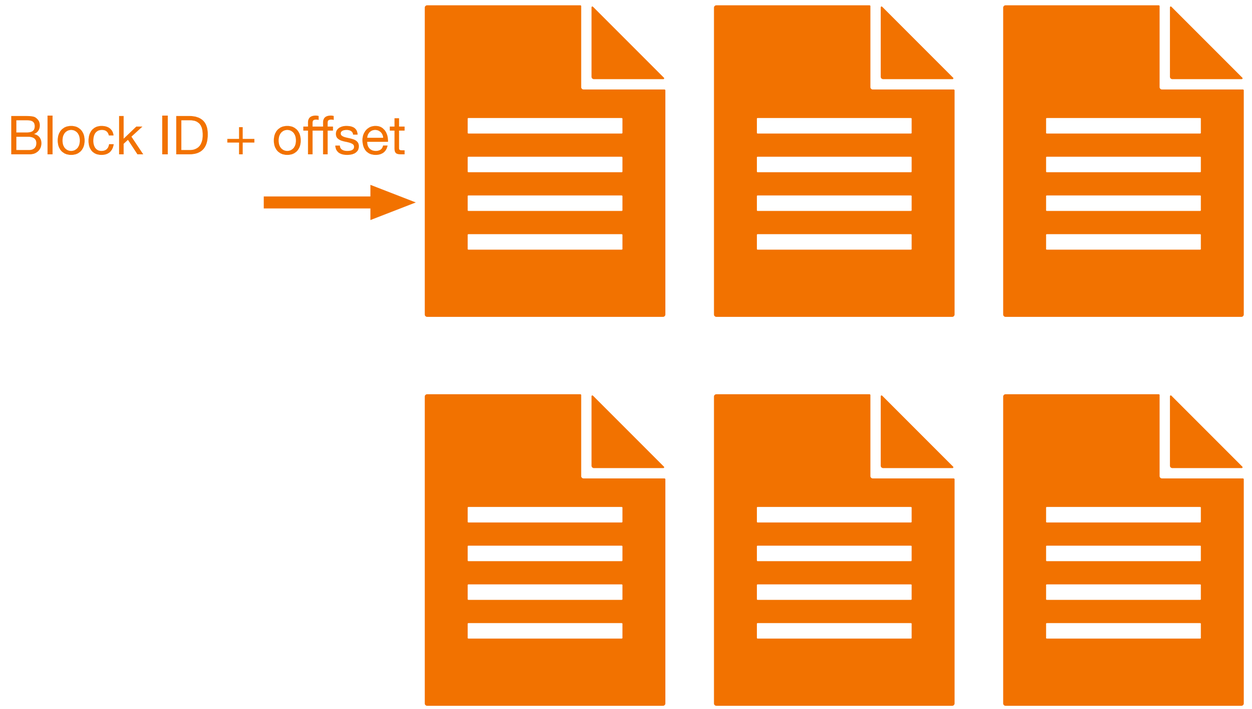
```
locle : sle x0, x0, x1;
        ret
```

***Can’t be implemented in Rec!***

# Asm and Rec

## *Memory Model*



# main

```
fn main() {
  local x[3];
  x[0] ← 1;
  x[1] ← 2;
  // x ↦ [1, 2, 0]
  memmove(x + 1, x + 0, 2);
  // x ↦ [1, 1, 2]
  print(x[1]); print(x[2])
  // print 1, then 2
}
```

# main

```
fn main() {
  local x[3];
  x[0] ← 1;
  x[1] ← 2;
  // x ↦ [1, 2, 0]
  memmove(x + 1, x + 0, 2);
  // x ↦ [1, 1, 2]
  print(x[1]); print(x[2])
  // print 1, then 2
}
```

# print

```
print : mov x8, PRINT;
        syscall; ret
```

# print

```
print : mov x8, PRINT;
        syscall; ret
```

***Can’t be implemented in Rec!***

# main

```
fn main() {
  local x[3];
  x[0] ← 1;
  x[1] ← 2;
  // x ↦ [1, 2, 0]
  memmove(x + 1, x + 0, 2);
  // x ↦ [1, 1, 2]
  print(x[1]); print(x[2])
  // print 1, then 2
}
```

# Our compiled program!

$$\texttt{onetwo} \triangleq \downarrow\texttt{main} \cup_a \downarrow\texttt{memmove} \cup_a \texttt{locle} \cup_a \texttt{print}$$

$\downarrow\texttt{prog}$ := compilation from **Rec** to **Asm**

$\texttt{prog1} \cup_a \texttt{prog2}$ := *syntactic linking operator* for **Asm**

# Our proof goal

$$\texttt{onetwo} \preceq_{\mathsf{S}} \texttt{onetwo}_{\mathsf{spec}}$$

1. Each step of the **`Asm`** program **`onetwo`** corresponds to zero or more steps of the **`Asm`**-level specification $\texttt{onetwo}_{\mathsf{spec}}$.
2. **`onetwo`** and $\texttt{onetwo}_{\mathsf{spec}}$ yield the same external behavior.

# From *Libraries* to *Modules*

$[\![\texttt{AsmLib}]\!]_{\mathsf{a}}$ is a *semantic module* for the *syntactic library* **`AsmLib`**.

A module $M \in Module(E)$ is a *labeled transition system* emitting events from a language-specific set of events $E$.

# An excerpt of module semantics

$$\textsc{ASM-INCOMING}\quad \frac{r(pc) = a \qquad a \in |A|}{(Wait, A) \xrightarrow{Jump?(r,m)}_a \{(Run(r, m), A)\}}$$

# Defining our proof goal

$$\texttt{onetwo} \;\preceq_{a,s}\; \texttt{onetwo}_{spec}$$

$$\triangleq$$

$$[\![\texttt{onetwo}]\!]_a \;\preceq\; [\![\texttt{onetwo}_{spec}]\!]_s$$

$M_1 \preceq M_2$ is the *simulation* between modules.

# Decomposing our program

$$[\![\texttt{onetwo}]\!]_a \preceq [\![\downarrow\texttt{main} \cup_a \downarrow\texttt{memmove} \cup_a \texttt{locle} \cup_a \texttt{print}]\!]_a$$

By definition of **onetwo**.

# Decomposing into linked **Asm** modules

$$[\![ {\downarrow}\mathbf{main} \cup_a {\downarrow}\mathbf{memmove} \cup_a \mathbf{locle} \cup_a \mathbf{print} ]\!]_a \preceq [\![ {\downarrow}\mathbf{main} ]\!]_a \oplus_a [\![ {\downarrow}\mathbf{memmove} ]\!]_a \oplus_a [\![ \mathbf{locle} ]\!]_a \oplus_a [\![ \mathbf{print} ]\!]_a$$

$M_1 \oplus_a M_2$ is a *semantic linking operator* for **Asm**.

It synchronizes the **Jump** events of $M_1$ and $M_2$.

# Semantically linking `Asm` modules

$$M_1 \oplus_{\mathsf{a}} M_2$$

ASM-LINK-JUMP

$$\frac{(d' = \mathsf{L} \wedge \mathbf{r}(\mathbf{pc}) \in \mathbf{d}_1) \vee (d' = \mathsf{R} \wedge \mathbf{r}(\mathbf{pc}) \in \mathbf{d}_2) \vee (d' = \mathsf{E} \wedge \mathbf{r}(\mathbf{pc}) \notin \mathbf{d}_1 \cup \mathbf{d}_2) \qquad d \neq d'}{(d, \mathsf{None}, \mathbf{Jump}(\mathbf{r}, \mathbf{m})) \rightsquigarrow_{\mathbf{d}_1, \mathbf{d}_2} (d', \mathsf{None}, \mathbf{Jump}(\mathbf{r}, \mathbf{m}))}$$

# Decomposing into linked **Asm** modules

$$[\![ \downarrow \texttt{main} \cup_a \downarrow \texttt{memmove} \cup_a \texttt{locle} \cup_a \texttt{print} ]\!]_a \preceq [\![ \downarrow \texttt{main} ]\!]_a \oplus_a [\![ \downarrow \texttt{memmove} ]\!]_a \oplus_a [\![ \texttt{locle} ]\!]_a \oplus_a [\![ \texttt{print} ]\!]_a$$

$M_1 \oplus_a M_2$ is a *semantic linking operator* for **Asm**.

It synchronizes the **Jump** events of $M_1$ and $M_2$.

# Translating Asm modules back into Rec modules

$$[\![\downarrow \texttt{main}]\!]_a \oplus_a [\![\downarrow \texttt{memmove}]\!]_a \oplus_a [\![\texttt{locle}]\!]_a \oplus_a [\![\texttt{print}]\!]_a \preceq$$
$$\lceil [\![\texttt{main}]\!]_r \rceil_{r \rightleftharpoons a} \oplus_a \lceil [\![\texttt{memmove}]\!]_r \rceil_{r \rightleftharpoons a} \oplus_a [\![\texttt{locle}]\!]_a \oplus_a [\![\texttt{print}]\!]_a$$

$\lceil \cdot \rceil_{r \rightleftharpoons a}$ is a *semantic wrapper.*

It embeds Rec modules into Asm and translates between Rec and Asm events.

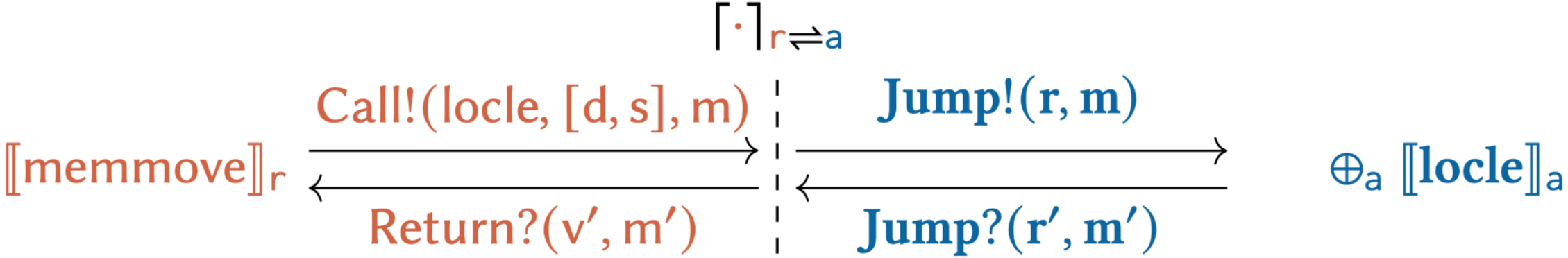
# Translating Asm modules back into Rec modules

$$[\![\downarrow \texttt{main}]\!]_a \oplus_a [\![\downarrow \texttt{memmove}]\!]_a \oplus_a [\![\texttt{locle}]\!]_a \oplus_a [\![\texttt{print}]\!]_a \preceq$$
$$\lceil [\![\texttt{main}]\!]_r \rceil_{r \rightleftharpoons a} \oplus_a \boxed{\lceil [\![\texttt{memmove}]\!]_r \rceil_{r \rightleftharpoons a} \oplus_a [\![\texttt{locle}]\!]_a} \oplus_a [\![\texttt{print}]\!]_a$$

$\lceil \cdot \rceil_{r \rightleftharpoons a}$ is a *semantic wrapper.*

It embeds Rec modules into Asm and translates between Rec and Asm events.

# The semantic wrapper $\lceil \cdot \rceil_{r \rightleftharpoons a}$ in use



# Translating Asm modules back into Rec modules

$$[\![\downarrow \texttt{main}]\!]_a \oplus_a [\![\downarrow \texttt{memmove}]\!]_a \oplus_a [\![\texttt{locle}]\!]_a \oplus_a [\![\texttt{print}]\!]_a \preceq$$
$$\lceil [\![\texttt{main}]\!]_r \rceil_{r \rightleftharpoons a} \oplus_a \lceil [\![\texttt{memmove}]\!]_r \rceil_{r \rightleftharpoons a} \oplus_a [\![\texttt{locle}]\!]_a \oplus_a [\![\texttt{print}]\!]_a$$

$\lceil \cdot \rceil_{r \rightleftharpoons a}$ is a *semantic wrapper.*

It embeds Rec modules into Asm and translates between Rec and Asm events.

# Abstracting `Asm` implementations

$$\lceil [\![\texttt{main}]\!]_{\texttt{r}} \rceil_{\texttt{r} \rightleftharpoons \texttt{a}} \oplus_{\texttt{a}} \lceil [\![\texttt{memmove}]\!]_{\texttt{r}} \rceil_{\texttt{r} \rightleftharpoons \texttt{a}} \oplus_{\texttt{a}} [\![\texttt{locle}]\!]_{\texttt{a}} \oplus_{\texttt{a}} [\![\texttt{print}]\!]_{\texttt{a}} \preceq$$
$$\lceil [\![\texttt{main}]\!]_{\texttt{r}} \rceil_{\texttt{r} \rightleftharpoons \texttt{a}} \oplus_{\texttt{a}} \lceil [\![\texttt{memmove}]\!]_{\texttt{r}} \rceil_{\texttt{r} \rightleftharpoons \texttt{a}} \oplus_{\texttt{a}} \lceil [\![\texttt{locle}_{\text{spec}}]\!]_{\text{s}} \rceil_{\texttt{r} \rightleftharpoons \texttt{a}} \oplus_{\texttt{a}} [\![\texttt{print}]\!]_{\texttt{a}}$$

`locle` can have a `Rec`-level specification because it has the same interaction behavior as a `Rec` program (i.e., no syscalls).

# Abstracting `Asm` implementations

$$\lceil [\![\texttt{main}]\!]_r \rceil_{r \rightleftharpoons a} \oplus_a \lceil [\![\texttt{memmove}]\!]_r \rceil_{r \rightleftharpoons a} \oplus_a \lceil [\![\texttt{locle}_{\text{spec}}]\!]_s \rceil_{r \rightleftharpoons a} \oplus_a [\![\texttt{print}]\!]_a \preceq$$
$$\lceil [\![\texttt{main}]\!]_r \rceil_{r \rightleftharpoons a} \oplus_a \lceil [\![\texttt{memmove}]\!]_r \rceil_{r \rightleftharpoons a} \oplus_a \lceil [\![\texttt{locle}_{\text{spec}}]\!]_s \rceil_{r \rightleftharpoons a} \oplus_a [\![\texttt{print}_{\text{spec}}]\!]_s$$

`print` makes a syscall, so it must have an `Asm`-level specification.

# Moving away from Asm

$$\lceil [\![\mathtt{main}]\!]_r \rceil_{r \rightleftharpoons a} \oplus_a \lceil [\![\mathtt{memmove}]\!]_r \rceil_{r \rightleftharpoons a} \oplus_a \lceil [\![\mathtt{locle}_{\mathsf{spec}}]\!]_s \rceil_{r \rightleftharpoons a} \oplus_a [\![\mathtt{print}_{\mathsf{spec}}]\!]_s \preceq$$
$$\lceil [\![\mathtt{main}]\!]_r \oplus_r [\![\mathtt{memmove}]\!]_r \oplus_r [\![\mathtt{locle}_{\mathsf{spec}}]\!]_s \rceil_{r \rightleftharpoons a} \oplus_a [\![\mathtt{print}_{\mathsf{spec}}]\!]_s$$

Semantic linking distributes over semantic wrappers.

# Semantic linking over semantic wrappers

REC-TO-ASM-LINK

$$\lceil M_1 \rceil_{r \rightleftharpoons a} \oplus_a \lceil M_2 \rceil_{r \rightleftharpoons a} \preceq \lceil M_1 \oplus_r M_2 \rceil_{r \rightleftharpoons a}$$

# Moving away from Asm

$$\lceil [\![\texttt{main}]\!]_r \rceil_{r \rightleftharpoons a} \oplus_a \lceil [\![\texttt{memmove}]\!]_r \rceil_{r \rightleftharpoons a} \oplus_a \lceil [\![\texttt{locle}_{\text{spec}}]\!]_s \rceil_{r \rightleftharpoons a} \oplus_a [\![\texttt{print}_{\text{spec}}]\!]_s \preceq$$
$$\lceil [\![\texttt{main}]\!]_r \oplus_r [\![\texttt{memmove}]\!]_r \oplus_r [\![\texttt{locle}_{\text{spec}}]\!]_s \rceil_{r \rightleftharpoons a} \oplus_a [\![\texttt{print}_{\text{spec}}]\!]_s$$

Semantic linking distributes over semantic wrappers.

# Shifting from semantic to syntactic linking

$$\lceil [\![\mathtt{main}]\!]_r \oplus_r [\![\mathtt{memmove}]\!]_r \oplus_r [\![\mathtt{locle}_{\mathsf{spec}}]\!]_s \rceil_{r \rightleftharpoons a} \oplus_a [\![\mathtt{print}_{\mathsf{spec}}]\!]_s \preceq$$
$$\lceil [\![\mathtt{main} \cup_r \mathtt{memmove}]\!]_r \oplus_r [\![\mathtt{locle}_{\mathsf{spec}}]\!]_s \rceil_{r \rightleftharpoons a} \oplus_a [\![\mathtt{print}_{\mathsf{spec}}]\!]_s$$

Syntactic linking and semantic linking are equivalent in **Rec**,
so we can link **main** and **memmove** semantically with $\cup_r$.

# Rec-level reasoning

$$\lceil [\![\mathbf{main} \cup_r \mathbf{memmove}]\!]_r \oplus_r [\![\mathbf{locle}_{\mathrm{spec}}]\!]_s \rceil_{r \rightleftharpoons a} \oplus_a [\![\mathbf{print}_{\mathrm{spec}}]\!]_s \preceq \lceil [\![\mathbf{main}_{\mathrm{spec}}]\!]_s \rceil_{r \rightleftharpoons a} \oplus_a [\![\mathbf{print}_{\mathrm{spec}}]\!]_s$$

$[\![\mathbf{main} \cup_r \mathbf{memmove}]\!]_r \oplus_r [\![\mathbf{locle}_{\mathrm{spec}}]\!]_s \preceq [\![\mathbf{main}_{\mathrm{spec}}]\!]_s$,
so we can use $[\![\mathbf{main}_{\mathrm{spec}}]\!]_s$ instead.

# locle spec

Pointers within the same block

Pointers in different blocks

# Rec-level reasoning

$$\lceil [\![\mathtt{main} \cup_r \mathtt{memmove}]\!]_r \oplus_r [\![\mathtt{locle}_{\mathrm{spec}}]\!]_s \rceil_{r \rightleftharpoons a} \oplus_a [\![\mathtt{print}_{\mathrm{spec}}]\!]_s \preceq \lceil [\![\mathtt{main}_{\mathrm{spec}}]\!]_s \rceil_{r \rightleftharpoons a} \oplus_a [\![\mathtt{print}_{\mathrm{spec}}]\!]_s$$

$[\![\mathtt{main} \cup_r \mathtt{memmove}]\!]_r \oplus_r [\![\mathtt{locle}_{\mathrm{spec}}]\!]_s \preceq [\![\mathtt{main}_{\mathrm{spec}}]\!]_s$,
so we can use $[\![\mathtt{main}_{\mathrm{spec}}]\!]_s$ instead.

# Asm-level reasoning

$$\lceil [\![\texttt{main}_{\text{spec}}]\!]_s \rceil_{r \rightleftharpoons a} \oplus_a [\![\texttt{print}_{\text{spec}}]\!]_s \preceq [\![\texttt{onetwo}_{\text{spec}}]\!]_s$$

We only need to make sure that the calls in `main` and the jumps in `print` line up to prove the simulation relation.

# Finished proof!

$$[\![\texttt{onetwo}]\!]_{\mathrm{a}} \preceq [\![\texttt{onetwo}_{\mathrm{spec}}]\!]_{\mathrm{s}}$$

The program's module refines its specification's module.

# Compiler correctness in DimSum

$$\textsc{compiler-correct}\ \frac{\downarrow R\ \textit{defined}}{[\![\downarrow R]\!]_a \preceq \lceil [\![R]\!]_r \rceil_{r \rightleftharpoons a}}$$

*Syntactic translation* via the compiler refines *semantic translation* via the wrapper.